MODULES

Angular apps are modular and Angular has its own modularity system called NgModules. Every Angular app has at least one NgModule class, the root module, conventionally named AppModule.

While the root module may be the only module in a small application, most apps have many more feature modules, each a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

An NgModule, whether a root or feature, is a class with an @ NgModule decorator.

> Decorators are functions that modify JavaScript classes. Angular has many decorators that attach metadata to classes so that it knows what those classes mean and how they should work

NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

- declarations - the view CLASSES that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.
- exports - the subset of declarations that should be visible and usable in the component templates of other modules.
- imports - other modules whose exported classes are needed by component templates declared in this module.
- providers - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.
- bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property

APP.MODULE.TS FILE

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
```

```
export class AppModule { }
```

Launch an application by bootstrapping its root module. During development you're likely to bootstrap the AppModule in a `main.ts` file like this one.

MAIN.TS FILE

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

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

NGMODULES VS. JAVASCRIPT MODULES

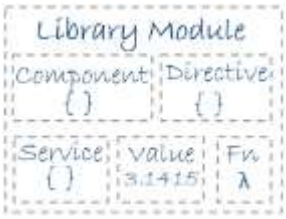
The `NgModule` is a class decorated with `@NgModule` which is a fundamental feature of Angular.

JavaScript also has its own module system for managing collections of JavaScript objects. It's completely different and unrelated to the NgModule system.

In JavaScript each file is a module and all objects defined in the file belong to that module. The module declares some objects to be public by marking them with the `export` key word. Other JavaScript modules use import statements to access public objects from other modules.

```
import { NgModule }     from '@angular/core';
import { AppComponent } from './app.component';
export class AppModule { }
```

ANGULAR LIBRARIES



Angular ships as a collection of JavaScript modules. You can think of them as library modules. Each Angular library name begins with the `@angular` prefix. You install them with the **npm** package manager and import parts of them with JavaScript `import` statements.

For example, import Angular's **Component** decorator from the `@angular/core` library like this:

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

You also import NgModules from Angular libraries using JavaScript import statements:

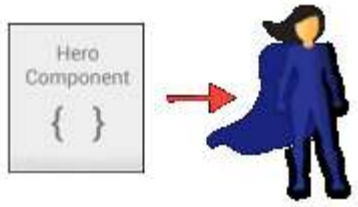
```
import { BrowserModule } from '@angular/platform-browser';
```

In the example of the simple root module above, the application module needs material from within that `BrowserModule`. To access that material, add it to the `@NgModule` metadata `imports` like this.

```
imports: [ BrowserModule ],
```

In this way you're using both the Angular and JavaScript module systems together. It's easy to confuse the two systems because they share the common vocabulary of "imports" and "exports".

COMPONENTS



A component controls a patch of screen called a view.

You define a component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.

For example, this `HeroListComponent` has a `heroes` property that returns an array of heroes that it acquires from a service. `HeroListComponent` also has a `selectHero()` method that sets a `selectedHero` property when the user clicks to choose a hero from that list.

```
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  constructor(private service: HeroService) { }
  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }
  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional **lifecycle hooks**, like `ngOnInit()` declared above.

TEMPLATES

You define a component's view with its companion **template**. A template is a form of HTML that tells Angular how to render the component.

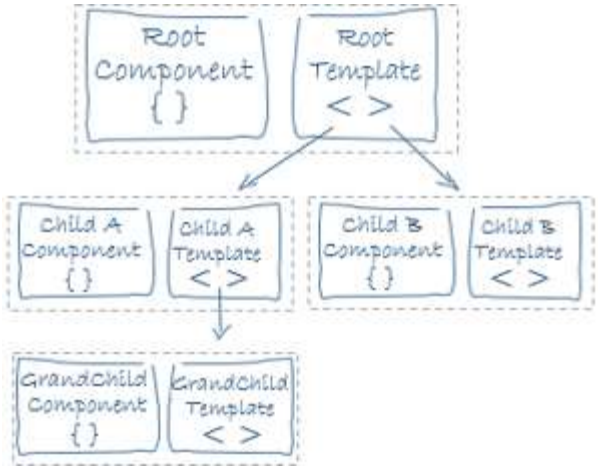
A template looks like regular HTML, except for a few differences. Here is a template for our `HeroListComponent`

```
<h2>Hero List</h2>


<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>


<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-
detail>
```

Although this template uses typical HTML elements like `<h2>` and `<p>`, it also has some differences. Code like `*ngFor`, `{{hero.name}}`, `(click)`, `[hero]`, and `<hero-detail>` uses Angular's template syntax.

In the last line of the template, the `<hero-detail>` tag is a custom element that represents a new component, `HeroDetailComponent`.

The `HeroDetailComponent` is a different component than the `HeroListComponent` you've been reviewing. The `HeroDetailComponent` (code not shown) presents facts about a particular hero, the hero that the user selects from the list presented by the `HeroListComponent`. The `HeroDetailComponent` is a **child** of the `HeroListComponent`.



Notice how <hero-detail> rests comfortably among native HTML elements. Custom components mix seamlessly with native HTML in the same layouts

METADATA

Metadata tells Angular how to process a class. Looking back at the code for HeroListComponent, you can see that it's just a class. There is no evidence of a framework, no "Angular" in it at all. In fact, HeroListComponent really is just A CLASS. It's not a component until you tell Angular about it.

To tell Angular that HeroListComponent is a component, attach **metadata** to the class. In TypeScript, you attach metadata by using a **decorator**. Here's some metadata for HeroListComponent

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

Here is the @Component decorator, which identifies the class immediately below it as a component class. The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

Here are a few of the most useful @Component configuration options:

- selector: CSS selector that tells Angular to create and insert an instance of this component where it finds a <hero-list> tag in parent HTML. For example, if an app's HTML contains <hero-list></hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.

- `templateUrl`: module-relative address of this component's HTML template
- `providers`: array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that the component's constructor requires a `HeroService` so it can get the list of heroes to display.

The metadata in the `@Component` tells Angular where to get the major building blocks you specify for the component. The template, metadata, and component together describe a view.

Apply other metadata decorators in a similar fashion to guide Angular behavior. `@Injectable`, `@Input`, and `@Output` are a few of the more popular decorators.

The architectural takeaway is that you must add metadata to your code so that Angular knows what to do

CREATING A COMPONENT

One of the big ideas behind Angular is the idea of components.

In our Angular apps, we write HTML markup that becomes our interactive application, but the browser only understands a limited set of markup tags; Built-ins like <select> or <form> or <video> all have functionality defined by our browser creator.

What if we want to teach the browser new tags? What if we wanted to have a `<weather>` tag that shows the weather? Or what if we want to create a `<login>` tag that shows a login panel?

This is the fundamental idea behind components: we will teach the browser new tags that have custom functionality attached to them.

Angular components are the basic building blocks of your app. Each component defines:

- Any necessary imports needed by the component
- A component decorator, which includes properties that allow you to define the template, CSS styling, animations, etc..
- A class, which is where your component logic is stored.

Angular components reside within the /src/app folder:

```
> src
  > app
    app.component.ts      // A component file
    app.component.html    // A component template file
```

```
  app.component.scss    // A component CSS file
  > about               // Additional component folder
  > contact             // Additional component folder
```

By default, the Angular CLI that we used to install the project just includes the /src/app/app.component files. We can use the CLI to generate a new component for us and you will see how the folder structure reacts.

THE BASIC COMPONENT FILE (APP.COMPONENT.TS)

Let's open up the /src/app/app.component.ts componen file that the CLI generated for us:

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'app';
}
```

As mentioned previously, at the top we have our imports, the component decorator in the middle (which defines the selector, template and style location), and the component class.

Notice the selector **app-root**? If you open /src/index.html you will see a custom HTML tag noted as `<app-root></app-root>`.

This is where that component will load! If you have `ng serve` running in the console, you will note that in the browser at http://localhost:4200, the HTML matches the /src/app/app.component.html file.

CREATING A NEW COMPONENT

The Angular CLI is used for more than just starting new projects. You can also use it to generate new components. In the console (you can open a second console so that your `ng serve` command is not halted), type:

```
> ng generate component home       or     > ng g c home


  // Output:
  create src/app/home/home.component.html (23 bytes)
  create src/app/home/home.component.spec.ts (614 bytes)
```

```
  create src/app/home/home.component.ts (262 bytes)
  create src/app/home/home.component.scss (0 bytes)
  update src/app/app.module.ts (467 bytes)
```

Let's look at the component code and then take these one at a time. Open our first TypeScript file: `src/app/home/home.component.ts`

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

@Component({
    selector: 'app-home',
    templateUrl: './home.component.html',
    styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
    constructor() { }
    ngOnInit() {
    }
}
```

Notice that we suffix our TypeScript file with `.ts` instead of `.js` The problem is our browser doesn't know how to interpret TypeScript files. To solve this gap, the `ng serve` command live-compiles our `.ts` to a `.js` file automatically

IMPORTING DEPENDENCIES

The `import` statement defines the modules we want to use to write our code. Here we're importing two things: `Component`, and `OnInit`.

We `import Component` from the module `"@angular/core"`. The `"@angular/core"` portion tells our program where to find the dependencies that we're looking for. In this case, we're telling the compiler that `"@angular/core"` defines and exports two JavaScript/TypeScript objects called `Component` and `OnInit`.

Similarly, we `import OnInit` from the same module. As we'll learn later, `OnInit` helps us to run code when we initialize the component. For now, don't worry about it.

Notice that the structure of this `import` is of the format `import { things } from wherever`. In the `{ things }` part what we are doing is called destructuring. Destructuring is a feature provided by ES6 and TypeScript. We will talk more about it in the next chapter.

The idea with `import` is a lot like `import` in Java or `require` in Ruby: we're pulling in these dependencies from another module and making these dependencies available for use in this file.

COMPONENT DECORATORS

After importing our dependencies, we are declaring the component:

```
@Component({
    selector: 'app-hello-world',
    templateUrl: './hello-world.component.html',
    styleUrls: ['./hello-world.component.css']
})
```

We can think of decorators as metadata added to our code. When we use `@Component` on the `HomeComponent` class, we are "decorating" `HomeComponent` class as a `Component`.

We want to be able to use this component in our markup by using a `<app-home>` tag. To do that, we configure the `@Component` and specify the `selector` as `app-home`.

```
@Component({
    selector: 'app-hello-world'
    // ... more here
})
```

The syntax of Angular's component selectors is similar to CSS selectors. For now, know that with this `selector` we're defining a new tag that we can use in our markup.

The `selector` property here indicates which DOM element this component is going to use. In this case, any `<app-home></app-home>` tags that appear within a template will be compiled using the `HomeComponent` class and get any attached functionality

ADDING A TEMPLATE WITH TEMPLATEURL

In our component we are specifying a `templateUrl` of `./home.component.html`. This means that we will load our template from the file `home.component.html` in the same directory as our component. Let's take a look at that file:

```
<p>
```

```
    Home works!
  </p>
```

Here we're defining a `p` tag with some basic text in the middle. When Angular loads this component it will also read from this file and use it as the template for our component.

ADDING A TEMPLATE

We can define templates two ways, either by using the `template` key in our `@Component` object or by specifying a `templateUrl`.

We could add a template to our `@Component` by passing the `template` option

```
@Component({
selector: 'app-hello-world',
      template: `
      <p>
      hello-world works inline!
      </p>
      `
})
```

Notice that we're defining our `template` string between backticks (` ... `). This is a new (and fantastic) feature of ES6 that allows us to do multiline strings. Using backticks for multiline strings makes it easy to put templates inside your code files.

ADDING CSS STYLES WITH STYLEURLS

Notice the key `styleUrls`:

```
styleUrls: ['./home.component.css']
```

This code says that we want to use the CSS in the file `home.component.css` as the styles for this component. Angular uses a concept called "style-encapsulation" which means that styles specified for a component only apply to that component.