ACCELERATING THROUGH
ANGULAR 2

Level 1

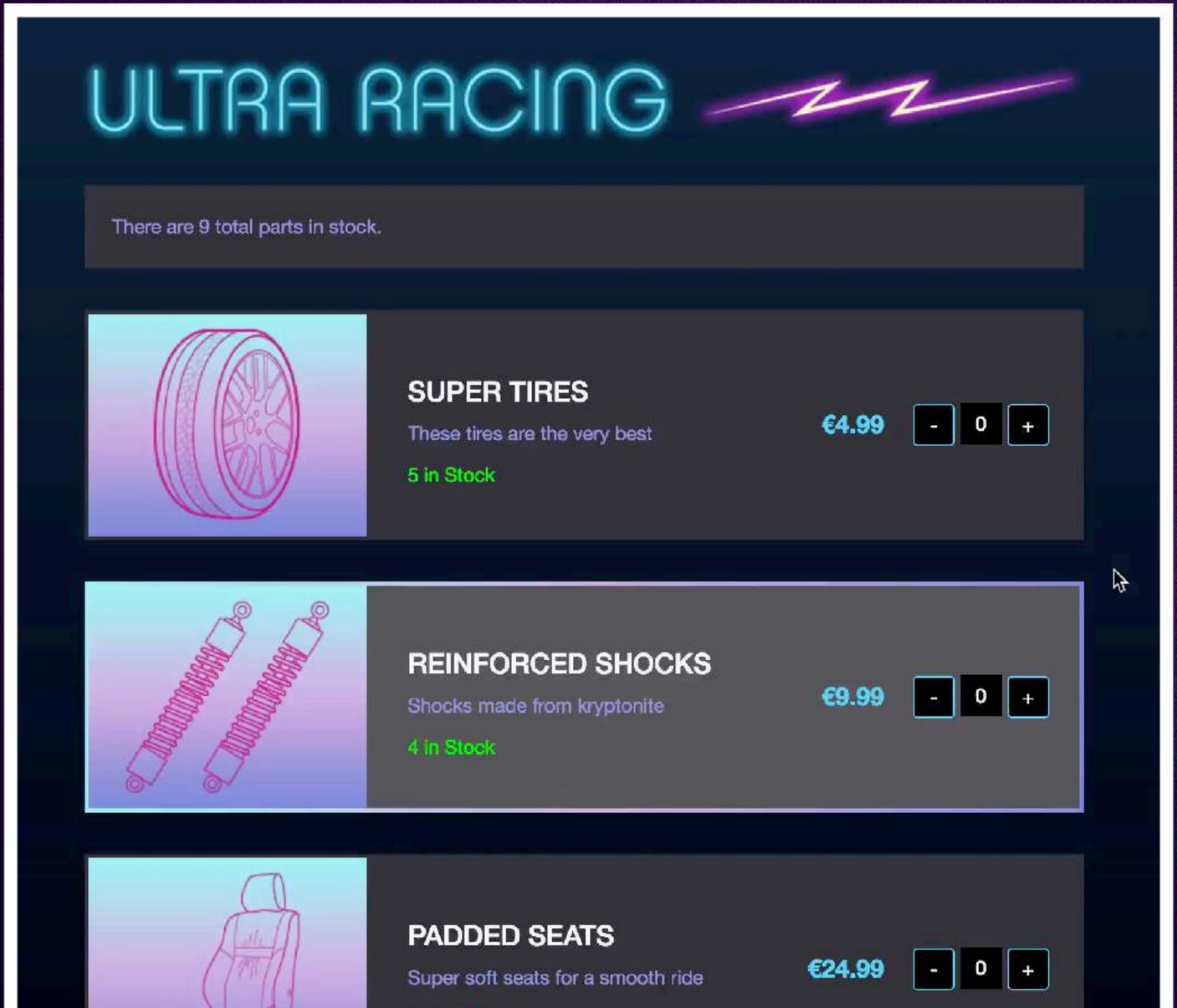
Our First Component

Section 1

What Is Angular?

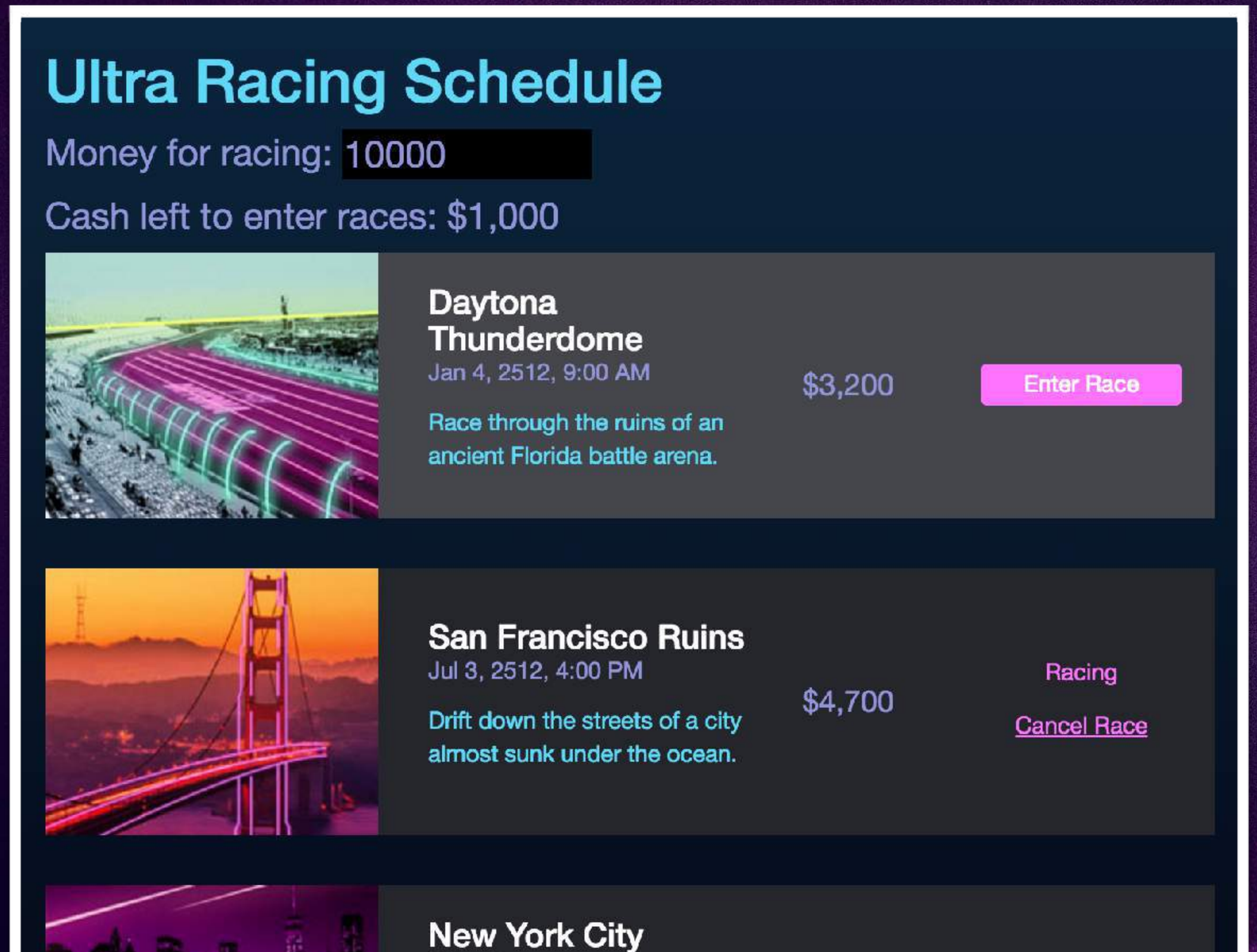
- Angular is a framework for dynamic web applications.
- Provides a way to organize your HTML, JavaScript, and CSS to keep your front-end code clean.
- Released in 2011.
- Mainly maintained by Google with the help of the open-source community.

What Will We Be Building in This Course?



ACCELERATING THROUGH ANGULAR 2

In the Challenges



What Will This Course Cover?

Level 1

Our First Component

Level 2

Structural Directives, Pipes & Methods

Level 3

Code Organization & Data Models

Level 4

Data Binding

Level 5

Services & HTTP

With lots of challenges between

What Is the Difference Between Angular 1 & 2?

Speed — Angular 2 is faster.

Components — Instead of controllers and scope, we use components, which feel simpler.

Simpler Directives — Creating custom directives is much simpler.

Intuitive Data Binding — When we need to link data to an HTML element or listen for a button clicking on the page, we have an intuitive syntax.

Services are now just a class.

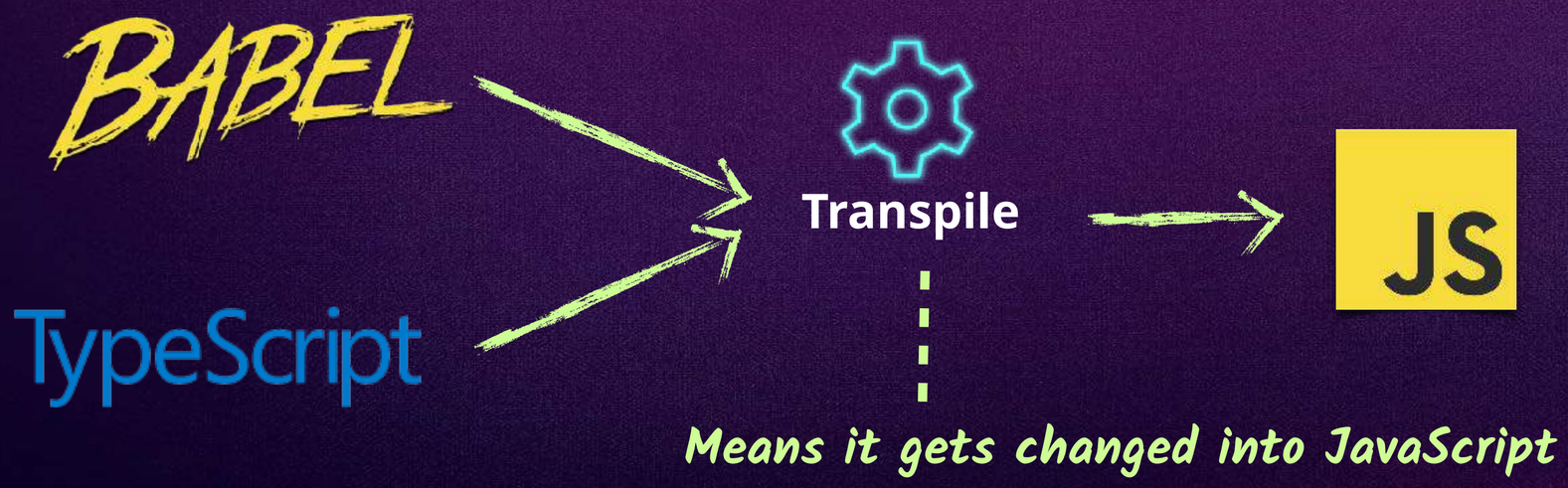
Many more small improvements.

What Language to Use With Angular 2?

JavaScript

But all browsers don’t support the newest version of JavaScript.

There are ways to access these features:



TypeScript: Our Language of Choice

TypeScript is Microsoft's extension of JavaScript that allows the use of all ES2015 features and adds powerful type checking and object-oriented features.

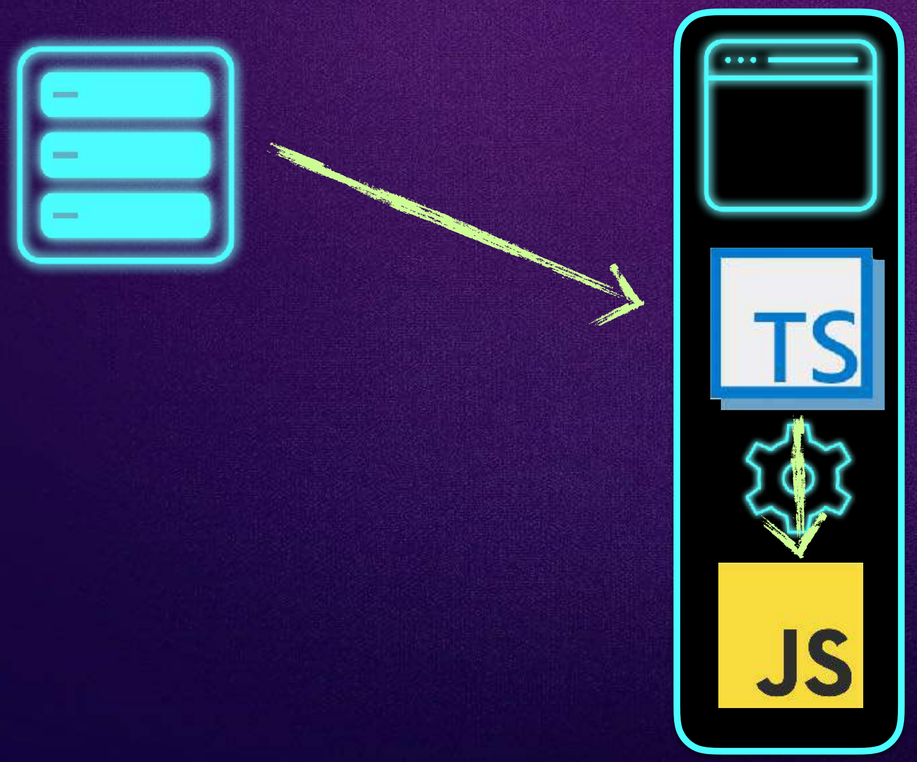
The Angular 2 source is programmed with TypeScript.

http://www.typescriptlang.org

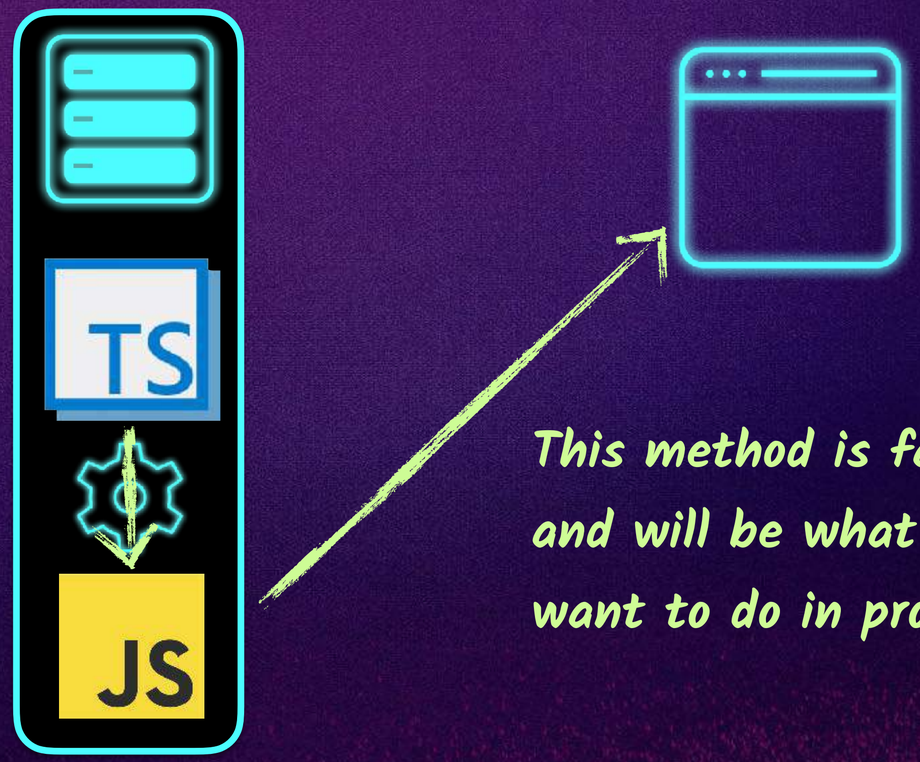
Transpiling Locations

Our browsers don't know how to read TypeScript out of the box, so we have two options when it comes to changing our TypeScript code into JavaScript.

Transpile to JavaScript in the browser



Transpile to JavaScript before shipping to browser



This method is faster and will be what you want to do in production.

Building Our Index

index.html HTML

```html
<!DOCTYPE html>
<html>

  <head>
    <!-- All the Angular 2 libraries needed -->
  </head>

</html>
```

We won't be covering all the libraries you need to load up Angular 2.

When you're ready to start developing, we suggest you go through the **5-minute QuickStart Guide**.

http://go.codeschool.com/angular2start

Creating Our First Custom Element

index.html HTML

```html
<!DOCTYPE html>
<html>

  <head>
    <!-- All the Angular 2 libraries needed -->
  </head>

  <body>
    <my-app>Loading App ...</my-app>
  </body>

</html>
```

This is where our Angular 2 application will load.

This could be named anything, even `<racing-app>`

Until our app gets loaded in the browser, we see:

Loading a JavaScript File Using SystemJS

index.html HTML

```html
<!DOCTYPE html>
<html>

  <head>
    <!-- All the Angular 2 libraries needed -->
    <script>
      ...
      System.import('app')
       .catch(function(err){ console.error(err); });
    </script>

  </head>

  <body>
    <my-app>Loading App ...</my-app>
  </body>

</html>
```

SystemJS is a JavaScript library that allows us to import other libraries.

app/main.ts

This loads our application's code.

Error messages should be printed out to the browser console.

ACCELERATING THROUGH ANGULAR 2

Writing Our First TypeScript File

app/main.ts TypeScript

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

Angular 2 library

`import` ES2015 feature used to import functions, objects, or primitives.

`Component` Function we will use to create our first Component.

Components are the basic building blocks of Angular 2 applications. A component controls a portion of the screen.

Component Is a Decorator Function

app/main.ts TypeScript

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

Our component decorator code goes here.

class AppComponent { }
```

A decorator adds more behavior to our class from outside the class.

It must be declared immediately before the class.

The decorator turns our plain old JavaScript class into a component.

Decorating Our First Component

app/main.ts TypeScript

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

@Component({
  selector: 'my-app',
  template: '<h1>Ultra Racing</h1>'
})
class AppComponent { }
```

index.html

```html
<body>
  <my-app>Loading App ...</my-app>
</body>
```

Often called metadata

`@Component`	Used to apply our component decorator to our class. *Decorators are a TypeScript feature.*
`selector`	The CSS selector for the HTML element where we want the component to load.
`template`	The content we want to load inside our selector.

Declaring the Root Angular Module

Modules are how we organize our application in Angular. Every Angular application must have a "root module," which we'll need to launch it.

app/main.ts

TypeScript

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

@Component({
  selector: 'my-app',
  template: '<h1>Ultra Racing</h1>'
})
class AppComponent { }

@NgModule({
  declarations: [ AppComponent ]
})
class AppModule { }
```

List of all components within the module

Dependencies to Render the Application

app/main.ts TypeScript

```typescript
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
...
```

`BrowserModule`	Module needed for running Angular websites.
`platformBrowserDynamic`	Angular library that will render the website.

This will allow us to bootstrap, or launch, the app.

Bootstrapping Our Component

Using our new dependencies.

app/main.ts

TypeScript

```
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

...

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

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

Loads required dependencies to launch our app in the browser

Indicates our root component

Render application using AppModule

Previewing our Rendered Application

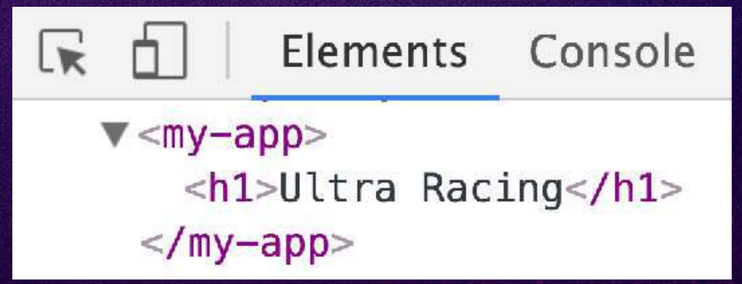
app/main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: '<h1>Ultra Racing</h1>'
})
class AppComponent { }


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

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

index.html

```
<body>
  <my-app>Loading App ...</my-app>
</body>
```

Viewing the Source



Our App Is Full of Components

Components are the building blocks of Angular 2 applications.

And they easily nest one inside the other.

Each component may have its own:

- *class file*
- *html file*
- *css file*

Sending Data Around

How do we send a property from our component class into our HTML?

app/main.ts TypeScript

```
...

@Component({
  selector: 'my-app',
  template: '<h1>???</h1>'
})
class AppComponent {
  title = 'Ultra Racing';
}

...
```

Inside a TypeScript class, we don't use the `var` or `let` keywords to declare class properties.

Though we do in regular methods.

Using Interpolation to Print Properties

Curly braces allow us to load in component properties — this is called interpolation.

app/main.ts TypeScript

```
...

@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>'
})
class AppComponent {
  title = 'Ultra Racing';
}

...
```

Loading an Object

What if we have an object we want to print out onto the screen?

app/main.ts TypeScript

```typescript
...
@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>'
})
class AppComponent {
  title = 'Ultra Racing';

  carPart = {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  };

}

...
```

ACCELERATING THROUGH ANGULAR 2

Template with Back Ticks

app/main.ts TypeScript

```typescript
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <h2>{{carPart.name}}<h2>
    <p>{{carPart.description}}<p>
    <p>{{carPart.inStock}} in Stock<p>`
})
class AppComponent {
  title = 'Ultra Racing';
  carPart = {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  };
}
```

Our template now uses back ticks instead of single quotes.

Single Quote

Back Tick

Using the back ticks allows us to have template strings, which allows us to be multiline.

This is another ES2015 feature.

What'd We Learn?

- Angular is a framework for dynamic web applications.
- We are coding Angular using TypeScript, a language that transpiles into JavaScript.
- NgModules group Angular code into blocks of functionality.
- Components are the basic building blocks of any Angular application.
- We use a custom HTML tag (aka, selector) to show where we want our component to load inside our HTML.
- Decorators are what turn our plain TypeScript classes into Components.

Level 2

Structural Directives

Section 1

Learning Angular Directives

A directive (within Angular) is how we add dynamic behavior to HTML.

There are three different kinds of directives:

Component

Has a template.

Structural

We will define these later.

Attribute

We won't get to these.

Looking Back at Our Code

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
   <h2>{{carPart.name}}<h2>
   <p>{{carPart.description}}<p>
   <p>{{carPart.inStock}} in Stock<p>`
})
class AppComponent {
  title = 'Ultra Racing';
  carPart = {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  };
}
```

What if we had more than one car part?

Adding an Array of Car Parts

main.ts TypeScript

```
...
})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  },
  {
    "id": 2,
    "name": "Reinforced Shocks",
    "description": "Shocks made from kryptonite",
    "inStock": 4
  }, { ... }];
}
```

Now that we have many car parts, how do we loop through each of these?

Adding an Array of Car Parts

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts">
        <h2>{{carPart.name}}<h2>
        <p>{{carPart.description}}<p>
        <p>{{carPart.inStock}} in Stock<p>
      </li>
    </ul>`
})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  }, { ... }, { ... }];
```

`*ngFor` is a structural directive.

`carPart` is a local variable.

`carParts` is the array to loop through.

The loop is run three times: once for each `carPart`.

Ultra Racing

- **Super Tires**

  These tires are the very best

  5 in Stock

- **Reinforced Shocks**

  Shocks made from kryptonite

  4 in Stock

- **Padded Seats**

  Super soft seats for a smooth ride

  0 in Stock

Learning Angular Directives

A directive (within Angular) is how we add dynamic behavior to HTML.

Component

Has a template.

Structural

Alters layout by adding, removing, or replacing HTML elements.

*ngFor

When Car Parts Are Out of Stock

When there are none in stock, how can we display “Out of Stock”?

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts">
        <h2>{{carPart.name}}<h2>
        <p>{{carPart.description}}<p>
        <p>{{carPart.inStock}} in Stock<p>
      </li>
    </ul>`
})
class AppComponent {
  ...
```

Should read “Out of Stock”

Using ngIf With a Conditional

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts">
        <h2>{{carPart.name}}<h2>
        <p>{{carPart.description}}<p>
        <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
        <p *ngIf="carPart.inStock === 0">Out of Stock</p>
      </li>
    </ul>`
})
class AppComponent {
  ...
```

If true, display this.

Ultra Racing

- **Super Tires**

  These tires are the very best

  5 in Stock

- **Reinforced Shocks**

  Shocks made from kryptonite

  4 in Stock

- **Padded Seats**

  Super soft seats for a smooth ride

  Out of Stock

`*ngIf` is another structural directive. It allows us to evaluate conditionals.

What'd We Learn?

- A directive (within Angular) is how we add dynamic behavior to HTML.
- A component directive has a template.
- A structural directive alters layout by adding, removing, or replacing HTML elements.

`*ngFor`	Loops through an array.
`*ngIf`	Shows content conditionally.

Level 2

Pipes & Methods

Section 2

Using Pipes to Format Screen Data

A pipe takes in data as input and transforms it to a desired output.

How can we write out car part names in capital letters?

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts">
        <h2>{{carPart.name | uppercase }}<h2>
        <p>{{carPart.description}}<p>
        <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
        <p *ngIf="carPart.inStock === 0">Out of Stock</p>
      </li>
    </ul>`
})
class AppComponent {
  ...
```

Similar to Linux pipe, if you're familiar with it.

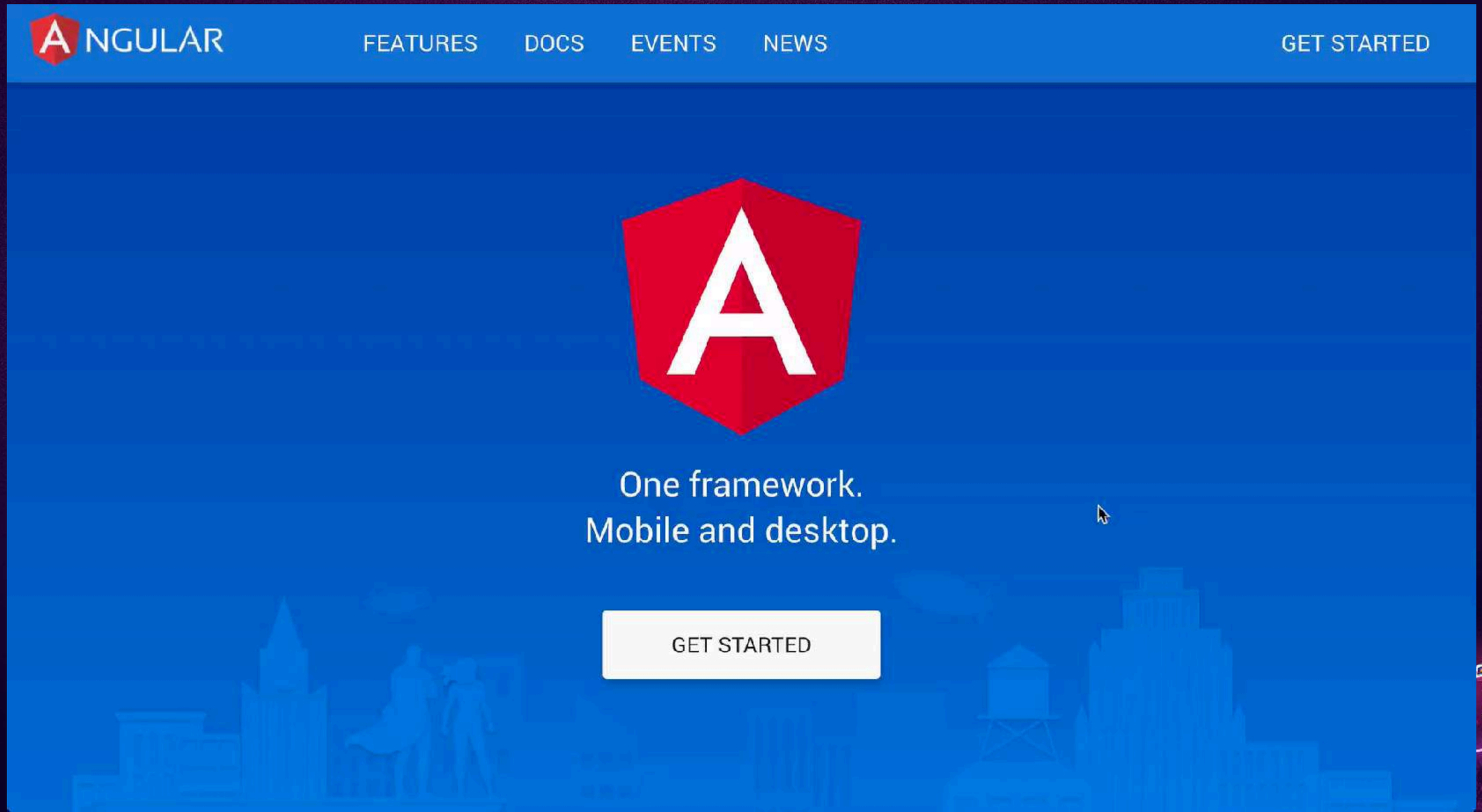
Adding a Price to Our Data

main.ts TypeScript

```
...
})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99
  },
  {
    "id": 2,
    "name": "Reinforced Shocks",
    "description": "Shocks made from kryptonite",
    "inStock": 4,
    "price": 9.99
  }, ...
```

But how do we format this properly?

Using Documentation to Look Up Pipes



Using the Currency Pipe With One Parameter

To format our currency, we use the ISO 4217 currency code.

Ex.: USD, EUR, or CAD

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts" >
        <h2>{{carPart.name | uppercase }}</h2>
        <p>{{carPart.description}}</p>
        <p>{{carPart.price | currency:'EUR'}}</p>
        <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
        <p *ngIf="carPart.inStock === 0">Out of Stock</p>
      <li>
    <ul>`
})
class AppComponent {
  ...
```

But we want the EUR symbol — how do we do that?

Using the Currency Pipe With Two Parameters

The second parameter is a boolean indicating if we should use the currency symbol.

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts" >
        <h2>{{carPart.name | uppercase }}</h2>
        <p>{{carPart.description}}</p>
        <p>{{carPart.price | currency:'EUR':true }}</p>
        <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
        <p *ngIf="carPart.inStock === 0">Out of Stock</p>
      <li>
    <ul>`
})
class AppComponent {
  ...
```

Notice the colon between parameters.

Additional Pipes to Use

`lowercase`	Well, lowercase...
`date`	Formats dates how you like them.
`number`	Formats numbers.
`decimal`	Formats decimals.
`replace`	Creates a new string, replacing specified characters.
`slice`	Creates a new list or string containing a subset of the elements.
`json`	Transforms any input to a JSON-formatted string.

You can also create custom pipes!

Great for debugging

Listing the Number of Car Parts in Stock

How could we display the total number of car parts in stock?

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts">
        ...
```

Modifying the Template

We'll add new code to our HTML template and print the result of a method we're about to define.

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <p>There are {{totalCarParts()}} total parts in stock.</p>
    <ul>
      <li *ngFor="let carPart of carParts">
        ...
```

We define this method inside of our component class.

Modifying the Template

Let's do the simplest thing and implement a class method that returns 10.

main.ts TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <p>There are {{totalCarParts()}} total parts in stock.</p>
  ` ...
})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];

  totalCarParts() {
    return 10;
  }
}
```

Inside a TypeScript class, we don't use the word "function," just like we don't use "let" to declare the properties.

ES2015 functionality enabled by TypeScript!

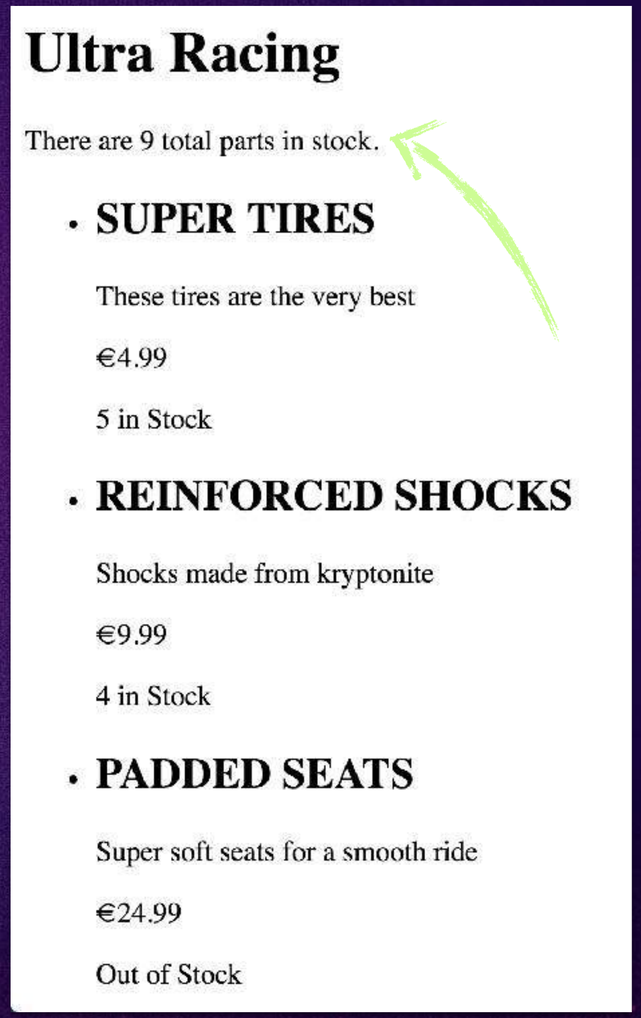
Implementing the Sum of Car Parts

Let's use an ES2015 `for of` **loop, like in our template.**

main.ts TypeScript

```
class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];

  totalCarParts() {
    let sum = 0;

    for (let carPart of this.carParts) {
       sum += carPart.inStock;
    }

    return sum;
  }
}
```

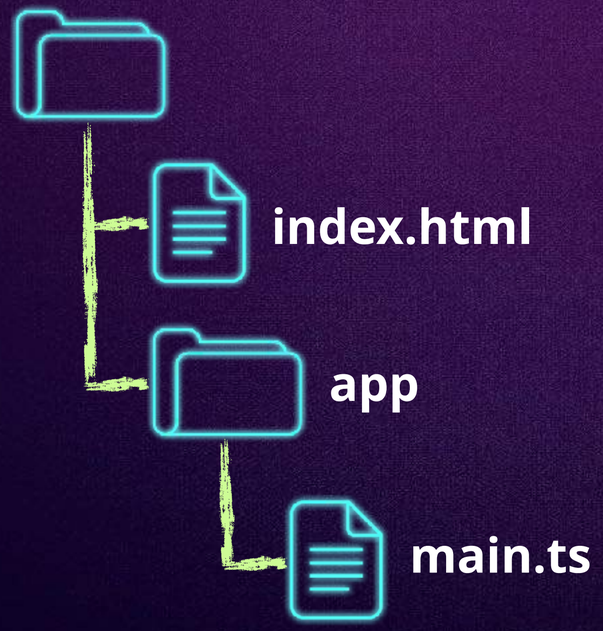
ES2015



What'd We Learn?

- We can use pipes to transform our template output.
- How to create and use methods in our components.

Our Current Application Structure



Bonus: Simplifying a Sum

```
totalCarParts() {
  let sum = 0;

  for (let carPart of this.carParts) {
     sum += carPart.inStock;
  }

  return sum;
}
```

Just for fun, let's go through a few ways we could simplify this code.

```
totalCarParts() {
  return this.carParts.reduce(function(prev, current) { return prev + current.inStock; }, 0 );
}
```

```
totalCarParts() {
  return this.carParts.reduce((prev, current) => prev + current.inStock, 0 );
}
```

The fat arrow (ES2015)

Level 3

Splitting to Two Components

Section 1

Splitting Things Into Pieces

We've been developing Angular 2 in one single file: main.ts**. This isn't going to scale, so let's split things up into pieces and get organized.**

main.ts

We will take our single file and split it into three.

main.ts
Where we'll bootstrap our app, loading our first component.

app.component.ts
This component contains our page header.

car-parts.component.ts
This contains our list of car parts.

After this, we will have two components instead of one.

Trimming Down Our main.ts

This is where we bootstrap our app, loading our first component.

main.ts TypeScript

```typescript
import { NgModule, Component } from ‘@angular/core’;
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from ‘@angular/platform-browser-dynamic';

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    ...`
})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];
  totalCarParts() { ... };
 }
...
```

There’s a bunch of code we need to move elsewhere.

Creating Our app.component.ts

We move most of our code into app.component.ts**.**

main.ts TypeScript

```
...


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

app.component.ts TypeScript

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

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    ...`
})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];
  totalCarParts() { ... };
}
```

However, this code is broken. We're bootstrapping our AppComponent, but we don't have access to this class.

How do we get access to a class from another file?

Exporting & Importing

We need to use the ES2015 feature of exporting and importing.

main.ts TypeScript

```
...
import { AppComponent } from './app.component';

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

app.component.ts TypeScript

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

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    ...`
})
export class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];
  totalCarParts() { ... };
}
```

First, we export the class we want to import.

Then, we import this class into our `main.ts`.

The names must be the same in each file.

One More File to Create

We need to create a car-parts.component.ts**.**

app

main.ts
Where we'll bootstrap our app, loading our first component.

app.component.ts
This component contains our page header.

car-parts.component.ts
This contains our list of car parts.

We need to split our AppComponent *into two components.*

Splitting Out Our Components

We need to remove the car parts-specific code from app.component.ts**.**

app.component.ts TypeScript

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

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <p>There are {{totalCarParts()}} total parts in stock.</p>
    <ul>...</ul>`
})
export class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];
  totalCarParts() { ... };
}
```

Splitting Out Our Car Parts Component

car-parts.component.ts TypeScript

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

@Component({
  selector: 'car-parts',
  template: `
    <p>There are {{totalCarParts()}} total parts in stock.</p>
    <ul>...</ul>`
})
export class CarPartsComponent {
  carParts = [...];
  totalCarParts() { ... };
}
```

Notice our new selector!

How do we tell our application about this component so it can be used?

Importing our Car Parts Component

main.ts TypeScript

```
import { AppComponent } from ‘./app.component';
import { CarPartsComponent } from './car-parts.component';

@NgModule({
  declarations: [
    AppComponent,
    CarPartsComponent
  ],
  imports: [ BrowserModule ],
  bootstrap: [ AppComponent ]
})
class AppModule { }
```

car-parts.component.ts TypeScript

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

@Component({
  selector: 'car-parts',
  template: `...`
})
export class CarPartsComponent {
  carParts = [...];
  totalCarParts() { ... };
}
```

Two things we need to do inside our `main.ts` file to make it work:

- Import our new component to *main.ts*
- Add CarPartsComponent to our module declarations array so it can be used through the rest of our application.

Using Our New Component

app.component.ts TypeScript

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

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
      <car-parts></car-parts>`
})
export class AppComponent {
  title = 'Ultra Racing';
}
```

Our new selector

Our app is rendering the CarPartsComponent by using the <car-parts> selector.

We've Separated Our Concerns!

app

main.ts
Where we'll bootstrap our app, loading our first component.

app.component.ts
This component contains our page header.

car-parts.component.ts
This contains our list of car parts.

And we've created our first reusable component.

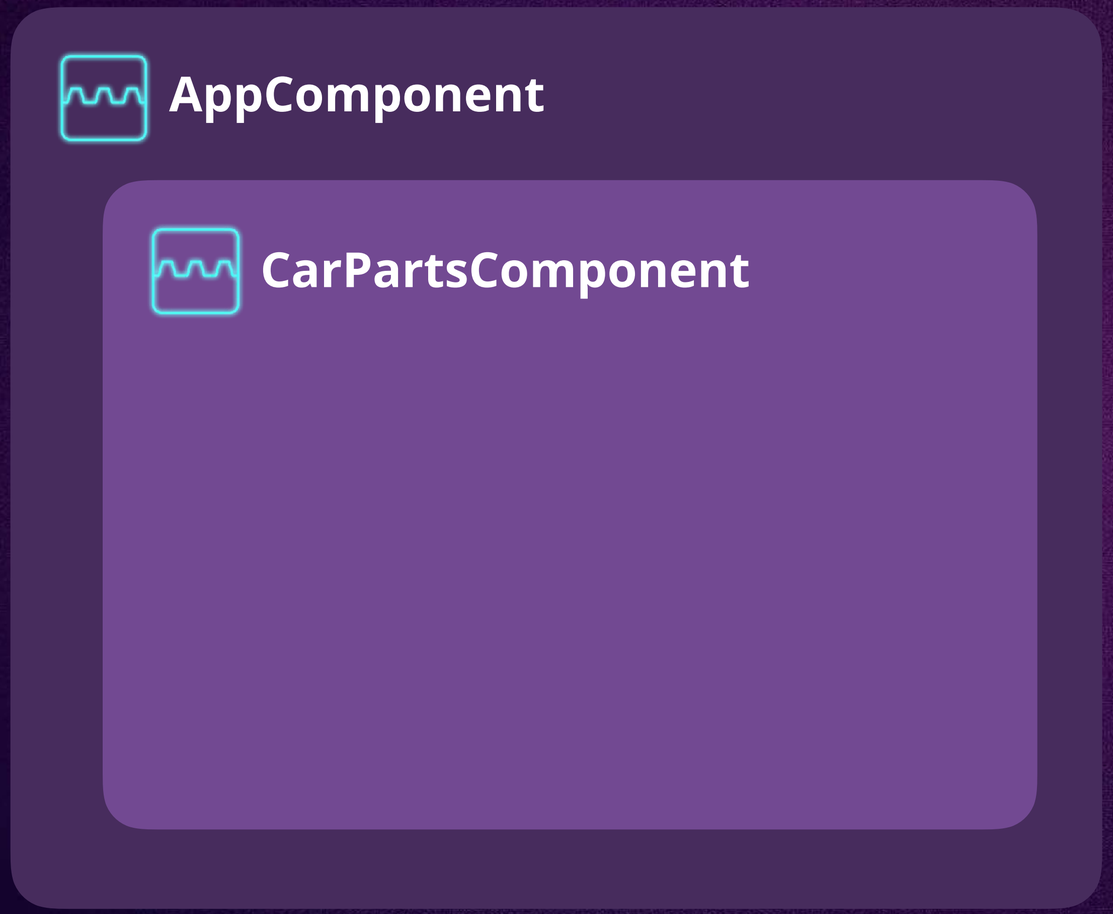
Components are meant to be reusable, so we could use them in different parts of our application.

Angular 2 Uses Component-based Architecture

Components can be all over the place in your application.

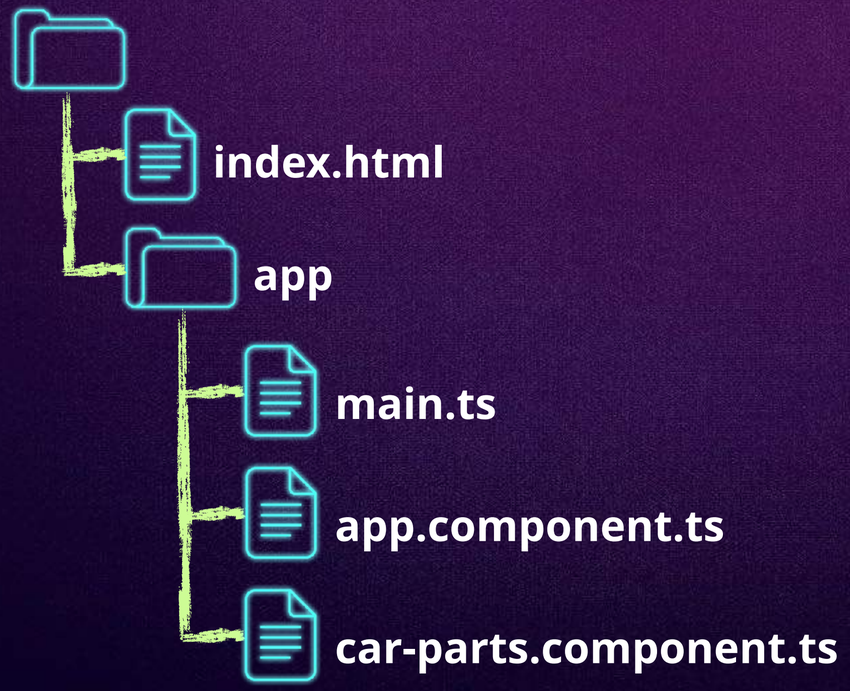
This isn't what we're building — we'll keep it even simpler.

Our Two Components



What'd We Learn?

- Our `main.ts` is where we import our first component and bootstrap it.
- In order to import a class, we must give it the `export` keyword.
- We use the `directives` metadata to list the directives our component uses.
- Components are the building blocks of our application.



Level 3

Component HTML & CSS

Section 2

How Do We Tie CSS to a Component?

car-parts.component.ts TypeScript

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

@Component({
  selector: 'car-parts',
  template: `...
              <p>{{carPart.description}}</p>
              <p>{{carPart.price | currency:'EUR':true}}</p>
            ...`

})
export class CarPartsComponent {
...
```

We have an HTML template, and we can even include CSS.

Adding Styles Array Metadata

car-parts.component.ts TypeScript

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

@Component({
  selector: 'car-parts',
  template: `...
    <p class="description">{{carPart.description}}</p>
    <p class="price">{{carPart.price | currency:'EUR':true}}</p>
            ...`,
  styles:[`
    .description {
      color: #444;
      font-size: small;
    }
    .price {
      font-weight: bold;
    }
  `]
})
export class CarPartsComponent {
```

New CSS classes

Notice this is an array.

Ultra Racing

There are 9 total parts in stock.

- **SUPER TIRES**

  These tires are the very best

  €4.99

  5 in Stock

- **REINFORCED SHOCKS**

  Shocks made from kryptonite

  €9.99

  4 in Stock

- **PADDED SEATS**

  Super soft seats for a smooth ride

  €24.99

  Out of Stock

The CSS Is Scoped to the Component

The HTML Source

```
<p _ngcontent-dcy-2 class="description">These tires are the very best</p>
<p _ngcontent-dcy-2 class="price">€4.99</p>
```

Notice the custom attribute.

The CSS Source

```
.description[_ngcontent-dcy-2] {
  color: #444;
  font-size: small;
}
.price[_ngcontent-dcy-2] {
    font-weight: bold;
}
```

Angular 2 adds this custom attribute to scope the CSS to only this component.

Kinda like properties are scoped, the CSS is scoped too!

Mind Blown

Splitting Things Into More Pieces

Up until now, we've included the HTML and CSS alongside our code.

car-parts.component.ts

I don't know about you, but this feels messy to me.

Let's split out our HTML and CSS into different files.

car-parts.component.html
Where our HTML for the component lives.

car-parts.component.css
Where our CSS for the component lives.

Our Current Component

car-parts.component.ts TypeScript

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

@Component({
  selector: 'car-parts',
  template: `
    <p>There are {{totalCarParts()}} total parts in stock.</p>
    <ul>...</ul>`,
   styles:[`
     .description {
       color: #444;
       font-size: small;
     }
     .price {
       font-weight: bold;
     }
   `]
})
export class CarPartsComponent {
  ...
```

We need to move out the HTML and CSS.

ACCELERATING THROUGH ANGULAR 2

Moving Out the HTML & CSS

car-parts.component.ts TypeScript

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

@Component({
  selector: 'car-parts',
  templateUrl: 'app/car-parts.component.html',
  styleUrls:['app/car-parts.component.css']
})
export class CarPartsComponent {
  ...
```

car-parts.component.html HTML

```html
<p>There are {{totalCarParts()}} total parts in stock.</p>
<ul>...</ul>
```

car-parts.component.css CSS

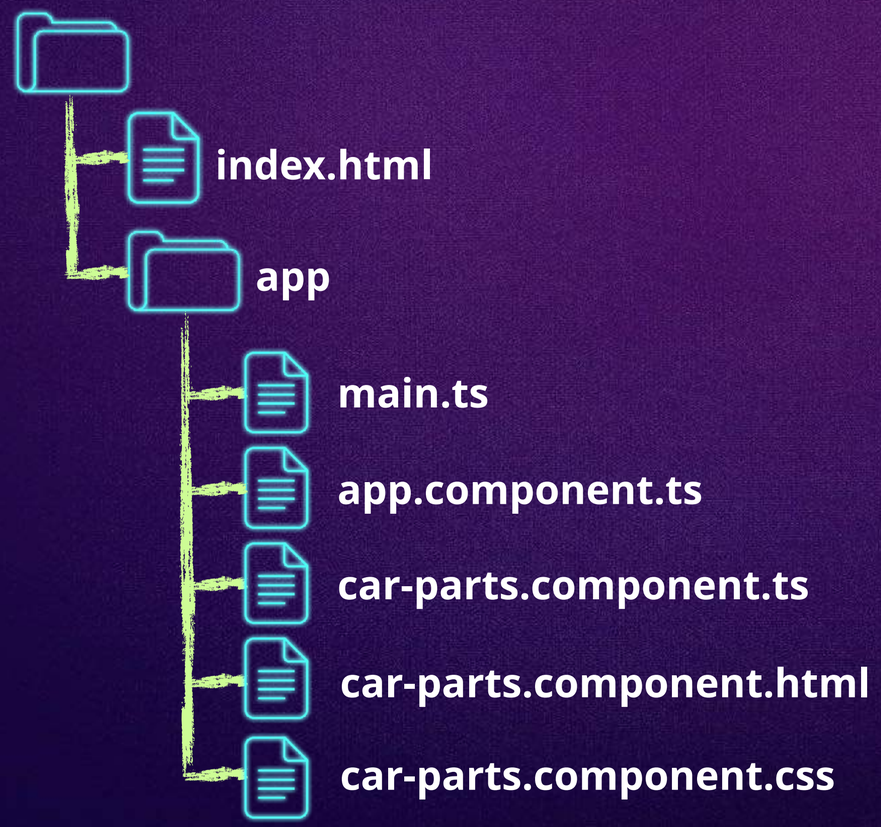
```css
.description {
  color: #444;
  font-size: small;
}
.price {
  font-weight: bold;
}
```

Once we create new files for our HTML and CSS, we can reference them inside our component metadata.

The CSS still gets scoped just like before.

What'd We Learn?

- We can include CSS just like we include our HTML template.
- CSS automatically gets scoped to our component.
- HTML and CSS can get split out into their own files.



Level 3

Mocks & Models

Section 3

Getting More Object Oriented

TypeScript gives us the ability to be more object oriented with our data, so let's create a model.

Which is basically a class in JavaScript.

car-part.ts TypeScript

```typescript
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
}
```

Notice we're declaring what type each of our properties are.
This is TypeScript.

This will allow our compiler to check our code and ensure we're getting and setting things properly.

Our Previous Code

car-parts.component.ts TypeScript

```
import { Component } from '@angular/core';
...
})
export class CarPartsComponent {
  carParts = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99
  }, { ... }, { ... }];
  ...
```

How do we use our new model?

car-part.ts TypeScript

```
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
}
```

Import the CarPart Model & Define Type

car-parts.component.ts TypeScript

```typescript
import { Component } from '@angular/core';
import { CarPart } from './car-part';
...
})
export class CarPartsComponent {
  carParts: CarPart[] = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99
  }, { ... }, { ... }];
  ...
```

Import the CarPart *model*

Tells TypeScript to treat this like an array of CarParts

car-part.ts TypeScript

```typescript
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
}
```

Nothing Else Needs to Change

car-parts.component.ts TypeScript

```
...
carParts: CarPart[] = [{
  "id": 1,
  "name": "Super Tires",
  "description": "These tires are the very best",
  "inStock": 5,
  "price": 4.99
}, { ... }, { ... }];
```

car-part.ts TypeScript

```
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
}
```

car-parts.component.html HTML

```
<p>There are {{totalCarParts()}} total parts in stock.</p>
<ul>
  <li *ngFor="let carPart of carParts" >
    <h2>{{carPart.name | uppercase}}</h2>
    <p class="description">{{carPart.description}}</p>
    <p class="price">{{carPart.price | currency:'EUR':true }}</p>
    <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
    <p *ngIf="carPart.inStock === 0">Out of Stock</p>
  </li>
</ul>
```

Cleaning Up Our Mock Data

Eventually we want to call out to a web service (API) to get the latest car parts, so it's a good practice to move our mock (fake) data out into its own file.

car-parts.component.ts TypeScript

```typescript
import { Component } from '@angular/core';
import { CarPart } from './car-part';
...
})
export class CarPartsComponent {
  carParts: CarPart[] = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99
  }, { ... }, { ... }];
  ...
```

mocks.ts TypeScript

Let's create a new file and move our mock data here.

Using Our Mock Data

car-parts.component.ts TypeScript

```typescript
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';

...
})
export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    this.carParts = CARPARTS;
  }
  ...
```

mocks.ts TypeScript

```typescript
import { CarPart } from './car-part';

export const CARPARTS: CarPart[] = [{
  "id": 1,
  "name": "Super Tires",
  "description": "These tires are the very best",
  "inStock": 5,
  "price": 4.99
}, { ... }, { ... }];
```

Notice we use `const` instead of `let` — this is an ES2015 feature that makes sure CARPARTS can't be reassigned.

`ngOnInit` **is invoked after the component is constructed and is the best place to initialize property values.**

We could have initialized in the constructor, but that'd be harder to test.

Yay, It Still Works!

Ultra Racing

There are 9 total parts in stock.

- **SUPER TIRES**

  These tires are the very best

  €4.99

  5 in Stock

- **REINFORCED SHOCKS**

  Shocks made from kryptonite

  €9.99

  4 in Stock

- **PADDED SEATS**

  Super soft seats for a smooth ride

  €24.99

  Out of Stock

We didn’t add any more functionality, but our code became a lot easier to maintain and scale.

Our Component Architecture

index.html

Includes `<my-app>` and loads `main.ts`

app

main.ts

Imports and bootstraps our first component

app.component.ts

Loads the header and our subcomponent

car-parts.component.ts

car-part.ts

The data model

mocks.ts

The fake data

car-parts.component.html

car-parts.component.css

app/car-parts.component.ts

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';

@Component({
  selector: 'car-parts',
  templateUrl: 'app/car-parts.component.html',
  styleUrls:['app/car-parts.component.css']
})
export class CarPartsComponent {
  carParts: CarPart[];
  ...
```

What'd We Learn?

- In TypeScript, we can use classes to model our data.
- TypeScript helps us specify class property types that help our compiler ensure we're writing good code.
- It's a good practice to keep your mock data separate from your model and your components, in its own file.

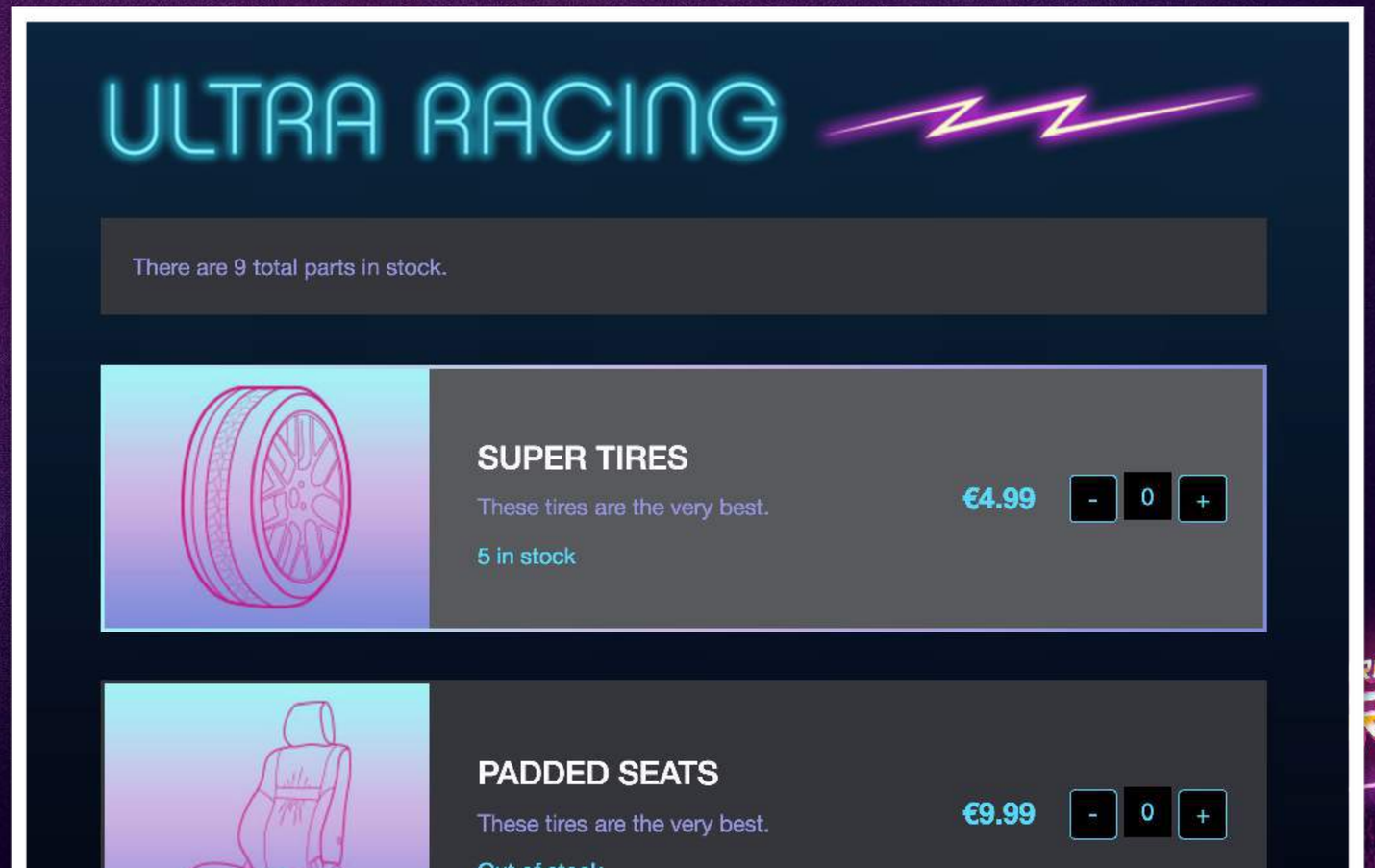
Level 4

Property & Class Binding

Section 1

Let’s Add Some Design

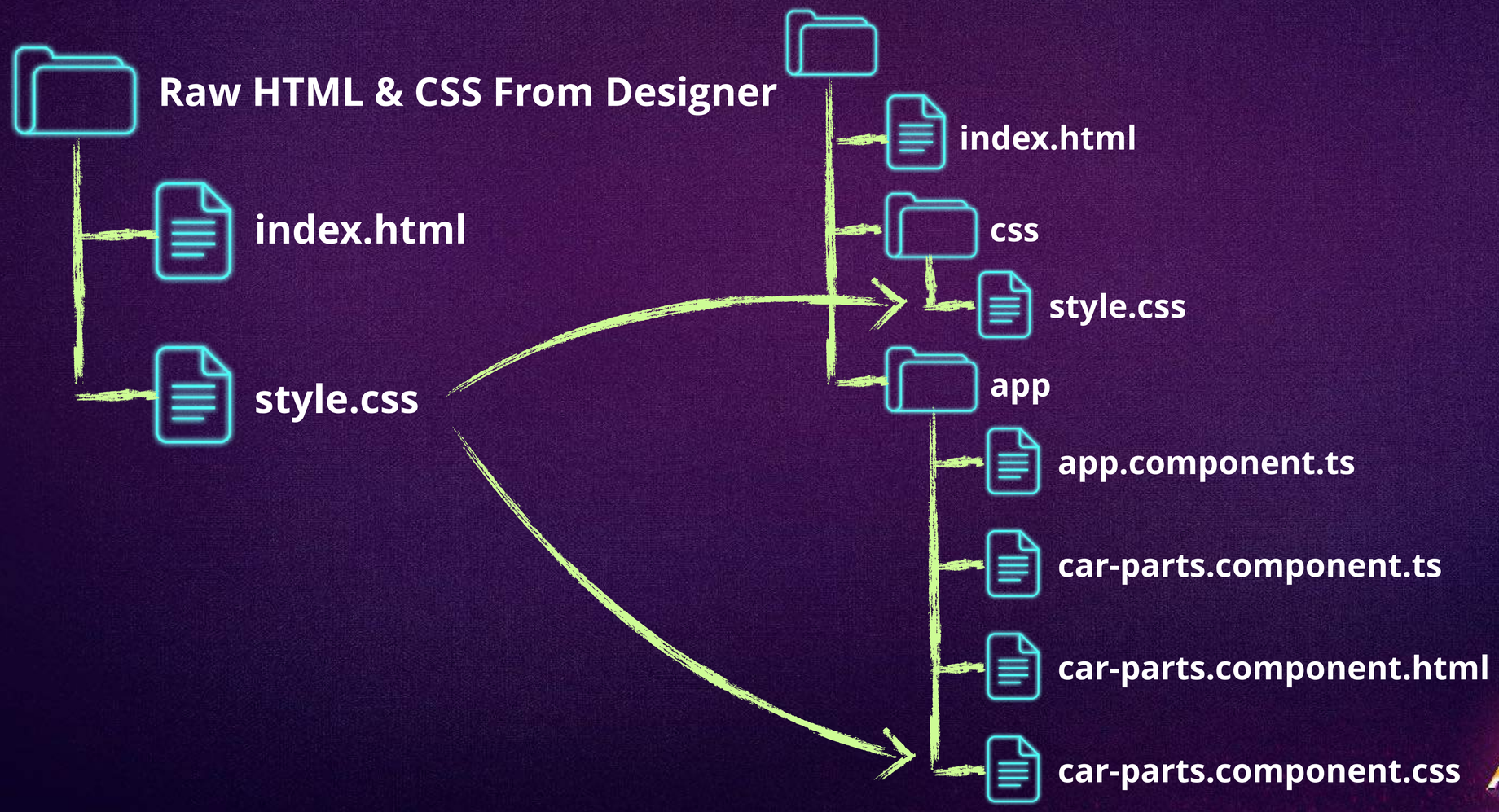
Not having to work with more complex HTML has been nice as we’ve learned Angular, but now we’re going to implement a better design.



We’ll be adding the images and quantity later this level.

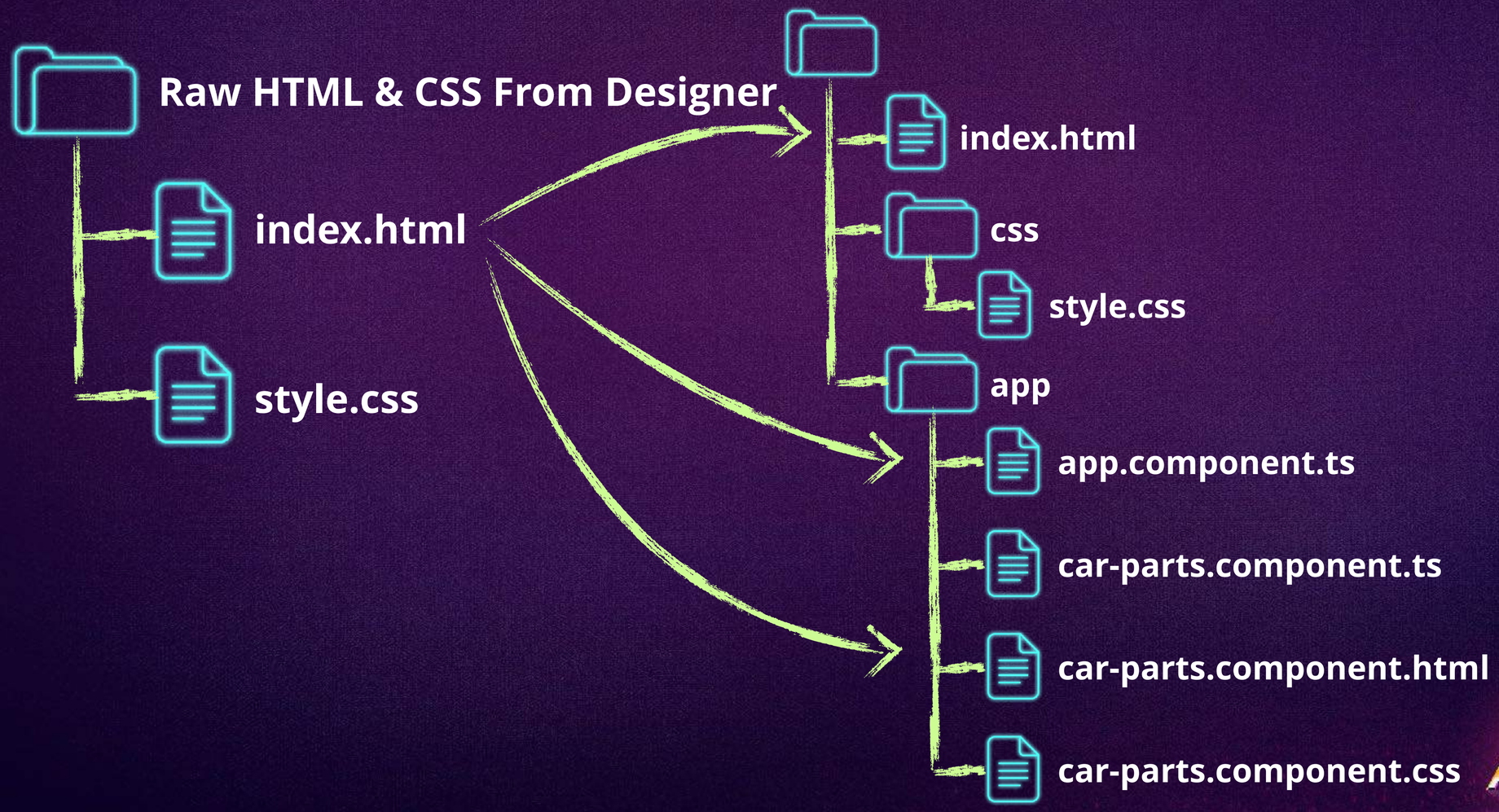
Moving the CSS

Styles get put in a new CSS folder and the car-parts.component.css **for styles specific to that component.**



Splitting Up the HTML

HTML gets split up into three files.



Our Current App

Much better, but let's figure out how to bring images in.

ULTRA RACING

There are 9 total parts in stock.

SUPER TIRES

These tires are the very best

5 in Stock

€4.99

REINFORCED SHOCKS

Shocks made from kryptonite

€9.99

The Ways Data Can Flow

When using a web framework like Angular that abstracts your code from HTML, there are a few different ways that data can flow.

JavaScript to HTML

Like we've been doing with properties from our components

HTML to JavaScript

Like a mouse click, hover, or key press

Both Ways

Like a text box, that should stay in sync

Note: We're saying JavaScript here because our TypeScript turns into JavaScript.

JavaScript to HTML

In our application thus far, we've been sending all sorts of data from our components into our HTML using interpolation.

car-parts.component.html TypeScript

```
<li class="card" *ngFor="let carPart of carParts" >
  <div class="panel-body">
    <table class="product-info">
      <tr>
        <td>
          <h2>{{carPart.name | uppercase}}</h2>
           <p class="description">{{carPart.description}}</p>
           <p class="inventory" *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
           <p class="inventory" *ngIf="carPart.inStock === 0">Out of Stock</p>
        </td>
        <td class="price">{{carPart.price | currency:'EUR':true }}</td>
      </tr>
    </table>
  </div>
</li>
```

When code is interpolated, the properties are read from the component and strings are printed.

So, how would we add an image tag with a dynamic image?

Adding Images to Our Model

We will add this property to our model, add new files, and add them to our mock data.

car-part.ts TypeScript

```typescript
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
  image: string;
}
```

mocks.ts TypeScript

```typescript
import { CarPart } from './car-part';

export let CARPARTS: CarPart[] = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99,
    "image": "/images/tire.jpg"
  },
  {
    "id": 2,
    "name": "Reinforced Shocks",
    "description": "Shocks made from kryptonite",
    "inStock": 4,
    "price": 9.99,
    "image": "/images/shocks.jpg"
  }, { ... } ];
```

Adding an Image to Our Template

We could try adding our image onto our page using interpolation.

car-parts.component.html TypeScript

```
<li class="card" *ngFor="let carPart of carParts" >
  <div class="panel-body">
    <div class="photo">
      <img src="{{carPart.image}}">
    </div>
    <table class="product-info">
      <tr>
        <td>
          <h2>{{carPart.name | uppercase}}</h2>
          ...
```

This would work just fine.

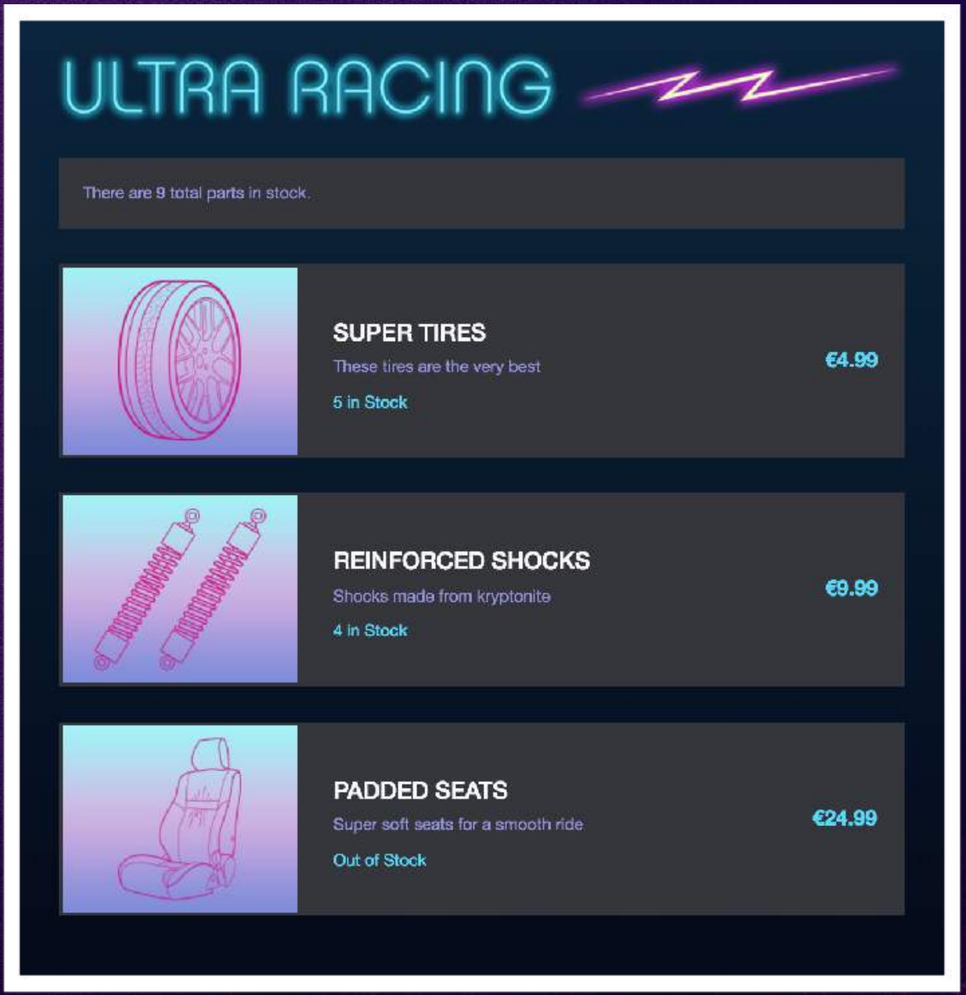
However, there's an alternative syntax we can use when we want to set DOM element property values.

Introducing Property Binding

Property binding allows us to glue component properties to DOM element properties.

car-parts.component.html TypeScript

```
<li class="card" *ngFor="let carPart of carParts" >
  <div class="panel-body">
    <div class="photo">
      <img [src]="carPart.image">
    </div>
    <table class="product-info">
      <tr>
        <td>
          <h2>{{carPart.name | uppercase}}</h2>
          ...
```



Notice the square brackets and no curly braces!

The square brackets tell Angular to set this DOM element property to our component property.

And if the component property changes, update this.

Additional Property Binding Examples

We can bind to any DOM element property. How do you think we could bind to these?

```
<div hidden>secret</div>
```

```
<button disabled>Purchase</button>
```

```
<img alt="Image Description">
```

Our previous solution

```
<img [src]="carPart.image">
```

Additional Property Binding Examples

All we need to do is add brackets and specify a component property.

```
<div [hidden]="!user.isAdmin">secret</div>
```

```
<button [disabled]="isDisabled">Purchase</button>
```

```
<img [alt]="image.description">
```

Our previous solution

```
<img [src]="carPart.image">
```

Adding a Featured Car Part

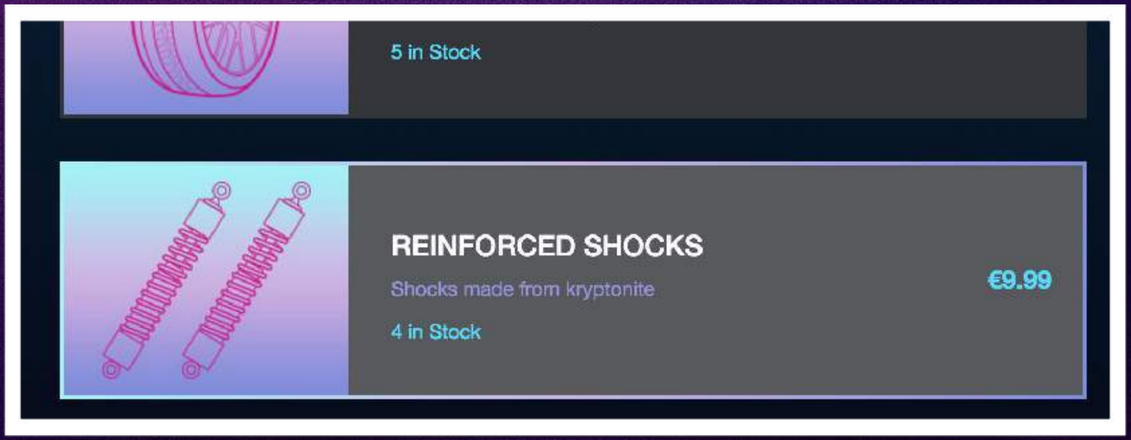
If a car part is marked as "featured," we want to add a class of featured **to it.**

car-parts.component.css CSS

```
...
.featured {
  background: #57595D;
  -webkit-border-image: -webkit-linear-gradient(right, #818fd8 0%, #cbb4e2 50%, #a6f2f5 100%);
       -o-border-image: linear-gradient(to left, #818fd8 0%, #cbb4e2 50%, #a6f2f5 100%);
          border-image: linear-gradient(to left, #818fd8 0%, #cbb4e2 50%, #a6f2f5 100%);
  border-image-slice: 1;
}
```

Here is the `featured` class, which adds a lighter background and a gradient border.

How do we add functionality to sometimes add this `featured` class?



Adding a Featured Property & Data

We need to add a new property to our car-part.ts **model and add mock data for it.**

car-part.ts TypeScript

```typescript
export class CarPart {
  ...
  image: string;
  featured: boolean;
}
```

mocks.ts TypeScript

```typescript
export let CARPARTS: CarPart[] = [{
    "id": 1,
    ...
    "featured": false
  },
  {
    "id": 2,
    ...
    "featured": true
  },
  {
    "id": 3,
    ...
    "featured": false
}];
```

Next, we need to conditionally add a class if this property is true.

Using a Class Property Binding

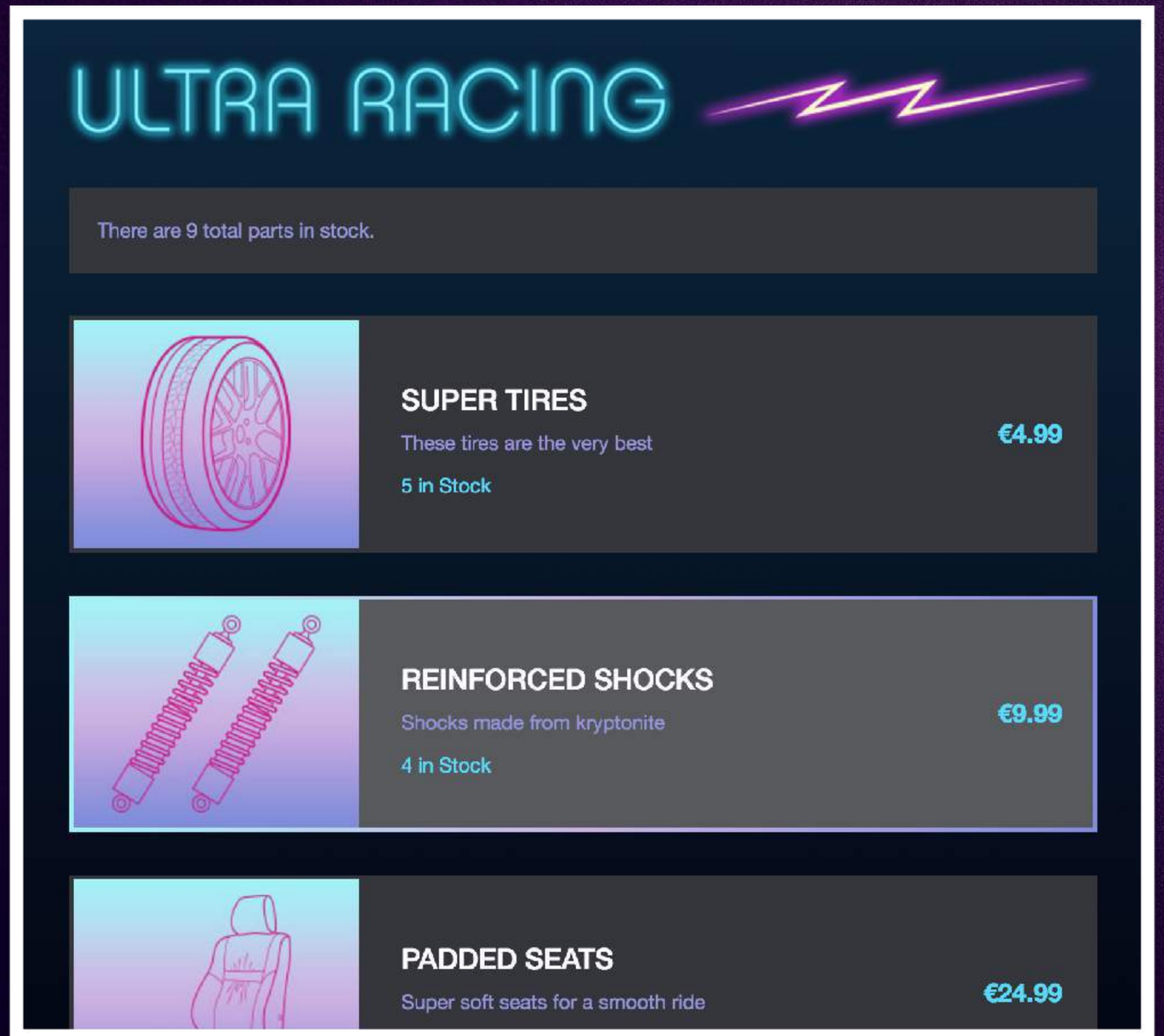
There's a unique syntax for binding to a class.

car-parts.component.html TypeScript

```
<ul>
  <li class="card" *ngFor="let carPart of carParts" [class.featured]="carPart.featured">
    <div class="panel-body">
      ...
    </div>
  </li>
</ul>
```

If `carPart.featured` is true, then the `featured` class is added. If `carPart.featured` is false, then the `featured` class is removed.

Looking Into the Web Page




```
[class.featured]="carPart.featured"
```

Looking at the source, we see that the element and the class are properly scoped.

```
<li _ngcontent-opf-2 class="card featured">
```

```
.featured[_ngcontent-opf-2] {
    background: #57595D;
    ...
}
```

How Not to Use Class Binding

You might be tempted to bind directly to the class element property:

```
<div [class]="property">
```

This will overwrite all classes.

```
<div [class.name]="property">
```

This will only add/remove the specific class.

Class names with dashes also work fine.

```
<div [class.my-name]="property">
```

What'd We Learn?

- Property binding allows us to bind component properties to any DOM element properties.
- Any update to the component property value will update the DOM property, but not vice versa — that's why it's "one-way binding."
- Class binding allows us to specify a CSS class to add to a DOM element if a component property is true.

Level 4

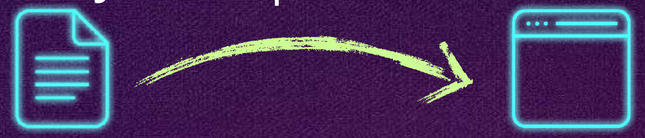
Event Binding

Section 2

Types of Data Binding

Property Binding
Class Binding

JavaScript to HTML

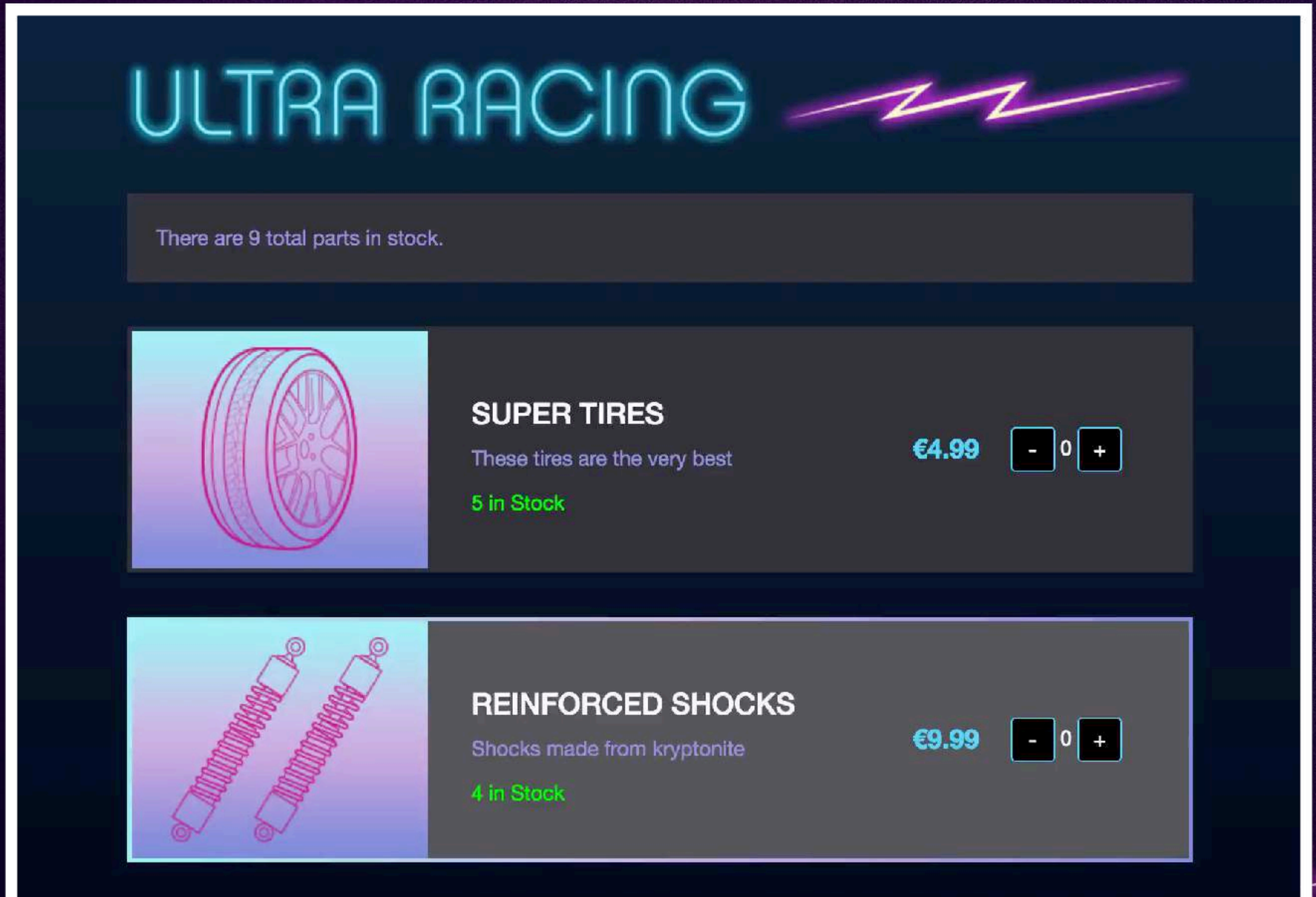


Event Binding

HTML to JavaScript

Like a mouse click, hover, or key press

Adding Events to Our Page



Adding a Quantity Property & Data

We need to add a new property to our car-part.ts **model and add mock data for it.**

car-part.ts TypeScript

```typescript
export class CarPart {
  ...
  image: string;
  featured: boolean;
  quantity: number;
}
```

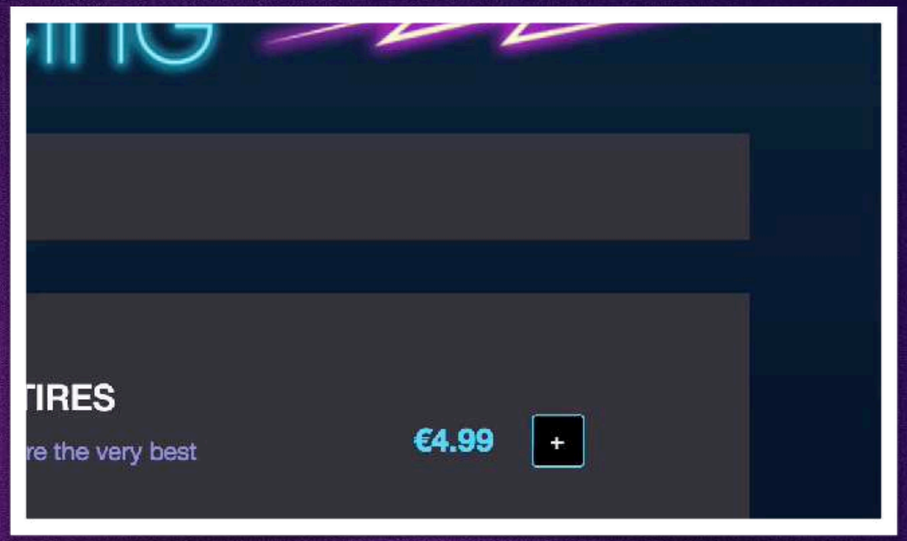
mocks.ts TypeScript

```typescript
export let CARPARTS: CarPart[] = [{
    "id": 1,
    ...
    "featured": false,
    "quantity": 0
  }, { ... }, { ... }];
```

Adding a Simple Button

car-parts.component.ts TypeScript

```
...
export class CarPartsComponent {
  ...

  upQuantity() {
    alert("You Called upQuantity");
  }
  ...
```



To capture an event from our template, we wrap the name of the event we want to listen to in parentheses and specify the method to call.

car-parts.component.html HTML

```
...
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
     <button class="increase" (click)="upQuantity()">+</button>
...
```

Making It Actually Work

Now let's use the carPart.quantity **that we have on each car part.**

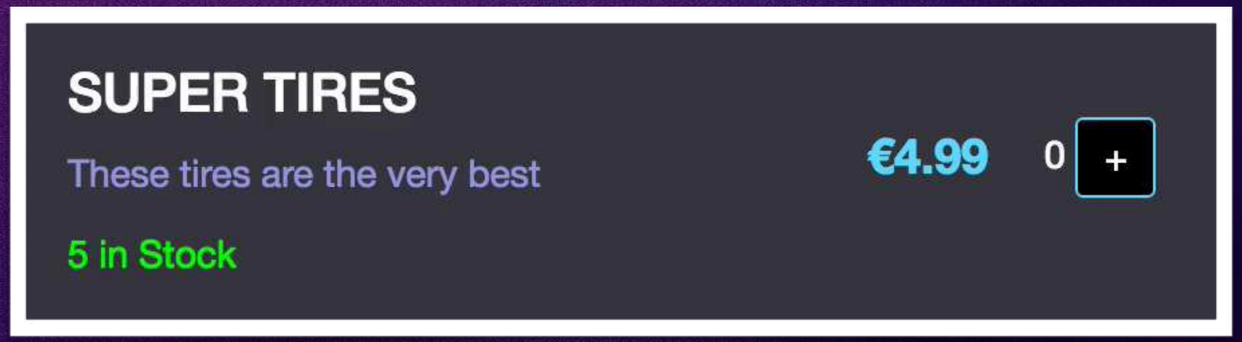
car-parts.component.html HTML

```html
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
    {{carPart.quantity}}
    <button class="increase" (click)="upQuantity(carPart)">+</button>
```

We need to send in the current carPart.

car-parts.component.ts TypeScript

```typescript
export class CarPartsComponent {
  ...

  upQuantity(carPart) {
    carPart.quantity++;
  }
```



Uh-oh — we can increase the quantity we want beyond what we have in stock.

Limited Incrementing

We shouldn't be able to add more quantity than we have in stock.

car-parts.component.html HTML

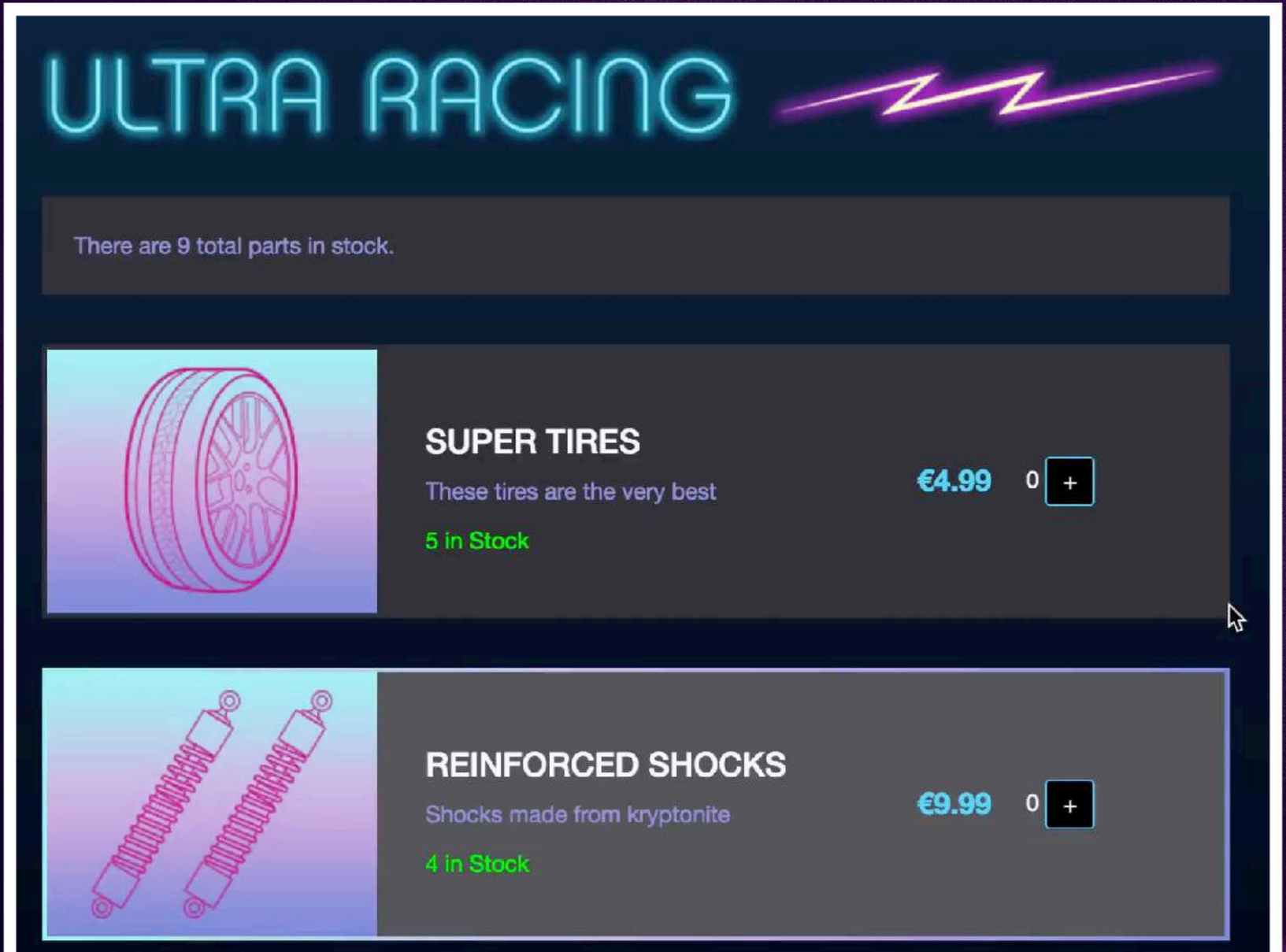
```html
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
     {{carPart.quantity}}
     <button class="increase" (click)="upQuantity(carPart)">+</button>
```

car-parts.component.ts TypeScript

```typescript
export class CarPartsComponent {
  ...

  upQuantity(carPart) {
    if (carPart.quantity < carPart.inStock) carPart.quantity++;
  }
```

Only add quantity if current quantity is less than we have in stock.

Now With Proper Limits



Adding Our Decrease Button

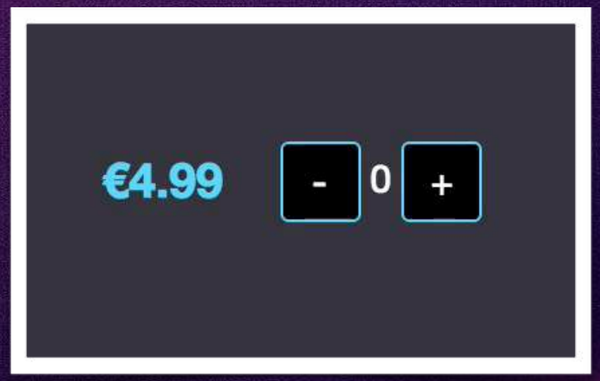
We should be able to decrease the quantity, but not below zero.

car-parts.component.html HTML

```html
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
    <button class="decrease" (click)="downQuantity(carPart)">-</button>
    {{carPart.quantity}}
    <button class="increase" (click)="upQuantity(carPart)">+</button>
```

car-parts.component.ts TypeScript

```typescript
export class CarPartsComponent {
  ...
  downQuantity(carPart) {
    if (carPart.quantity != 0) carPart.quantity--;
  }
}
```



Only subtract quantity if current quantity is not zero.

Other Events to Listen For

Any standard DOM event can be listened for by wrapping it in parentheses and removing the “on” at the beginning of the word.

```
<div (mouseover)="call()">
```

```
<input (blur)="call()">
```

```
<input (focus)="call()">
```

```
<input type="text" (keydown)="call()">
```

```
<form (submit)="call()">
```

Getting Additional Event Data

Sometimes you need additional event data, like which key is pressed or where the mouse is on the screen. This is what the Angular event object is for.

```
<input type="text" (keydown)="showKey($event)">
```

```
showKey(event) {
  alert(event.keyCode);
}
```

We can send the `$event` *object into our methods.*

```
<h2 (mouseover)="getCoord($event)">Hover Me</h2>
```

```
getCoord(event) {
  console.log(event.clientX + ", " + event.clientY);
}
```

We could also call `event.preventDefault();` to prevent a clicked link from being followed or a form from being submitted.

What'd We Learn?

- Event binding allows us to listen to any DOM event and call a component method when it's triggered.
- To listen to any event, we need to remove the "on" in front of the word, wrap it in parentheses, and specify a component method to call.
- If we need to access the event object, we can pass it in to our component method with `$event`.

Level 4

Two-way Binding

Section 3

Types of Data Binding

JavaScript to HTML

Property Binding
Class Binding

HTML to JavaScript

Event Binding

Both Ways

Like a text box, that should stay in sync

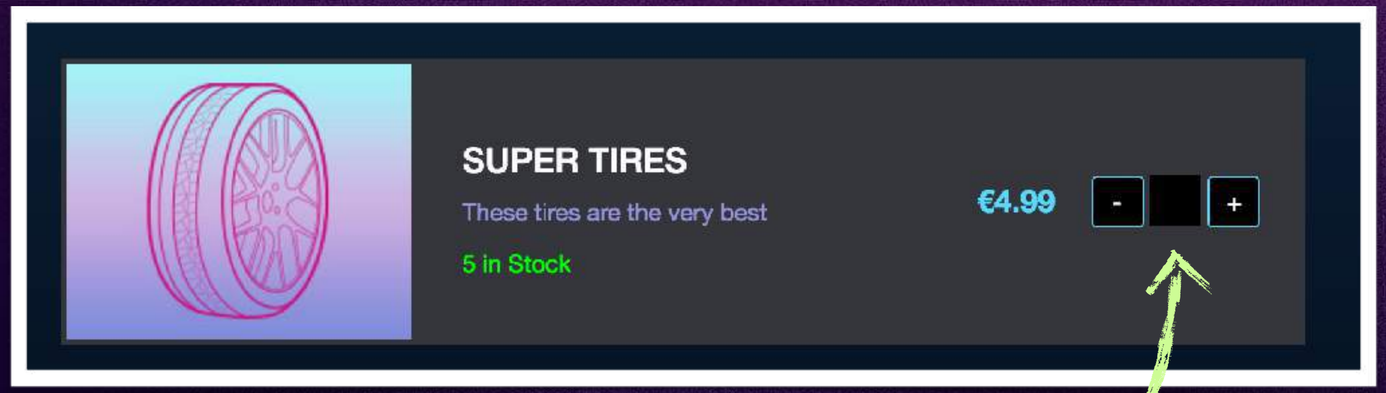
How can we bind properties from our component to our HTML, but also listen for events and keep things in sync?

Adding an Input Field

How can we allow for the user input of the quantity?

car-parts.component.html HTML

```html
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
     <button class="decrease" (click)="downQuantity(carPart)">-</button>
     <input class="number" type="text">
     <button class="increase" (click)="upQuantity(carPart)">+</button>
```



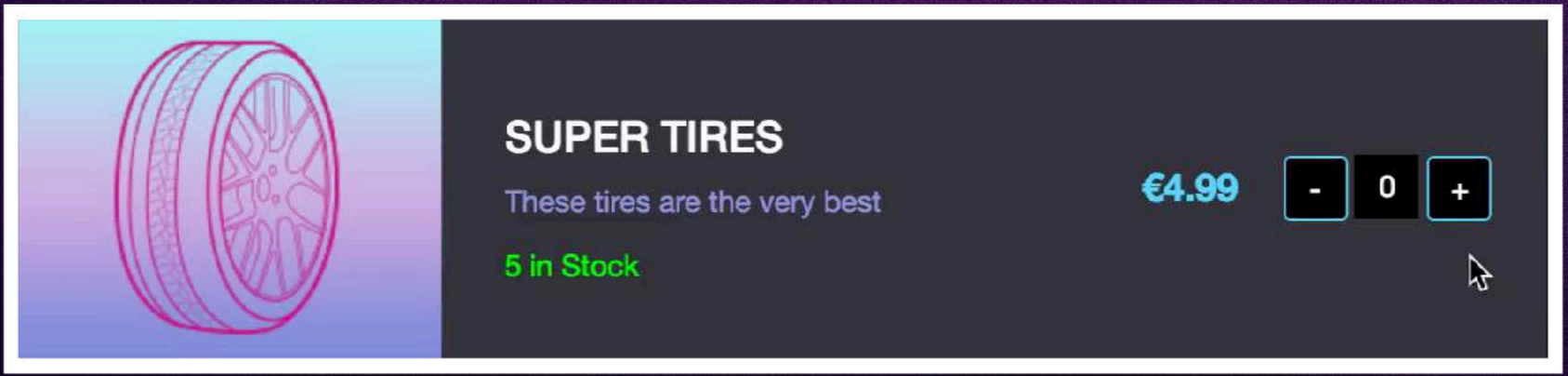
We should be able to adjust the quantity by typing into this field or by using the buttons.

Using Property Binding

The first thing we might try is to use property binding to bind the value to the quantity.

car-parts.component.html HTML

```html
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
     <button class="decrease" (click)="downQuantity(carPart)">-</button>
     <input class="number" type="text" [value]="carPart.quantity">
     <button class="increase" (click)="upQuantity(carPart)">+</button>
```



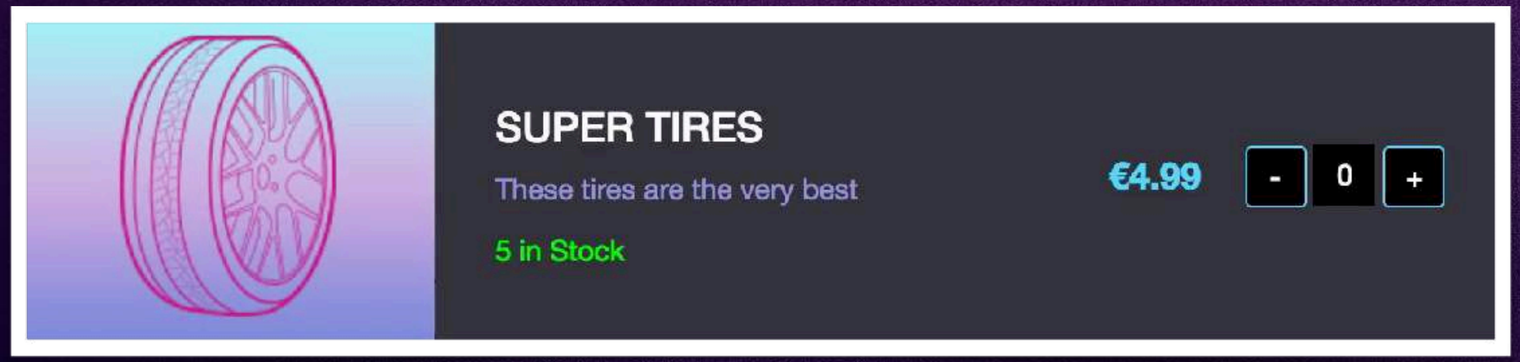
This gives us our quantity value in our input box, but only in one direction: from our component property to our input value.

Using Event Binding

We need to listen for the input event on our input box.

car-parts.component.html HTML

```html
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
     <button class="decrease" (click)="downQuantity(carPart)">-</button>
     <input class="number" type="text" [value]="carPart.quantity"
                                       (input)="carPart.quantity = $event.target.value">
     <button class="increase" (click)="upQuantity(carPart)">+</button>
```



Information is flowing two ways. *This works, but there's another way.*

Importing the FormsModule

Let's import the FormsModule **to get additional forms functionality into our codebase.**

main.ts TypeScript

```
...
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [ AppComponent ],
  imports: [ BrowserModule, FormsModule ],
  bootstrap: [ AppComponent ],
  providers: [ RacingDataService ],
})
class AppModule { }
...
```

Import FormsModule

Make form-specific functionality available to our whole app

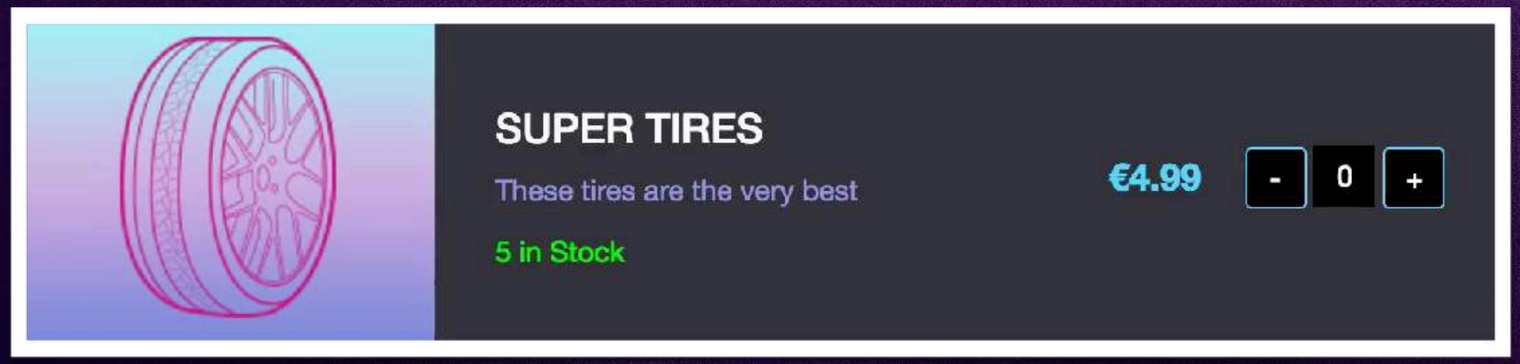
Using ngModel

ngModel **allows us to have one command to express two-way data binding.**

car-parts.component.html HTML

```html
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
     <button class="decrease" (click)="downQuantity(carPart)">-</button>
     <input class="number" type="text" [(ngModel)]="carPart.quantity" >
     <button class="increase" (click)="upQuantity(carPart)">+</button>
```

Notice that we're using both brackets and parentheses. [( )]



This syntax is sometimes called banana in a box — can you see why?

The ngModel Syntax

When we use the ngModel **syntax, we can only set it equal to a data bound property.**

```
[(ngModel)]="<must be data property>"
```

We will mostly use this for form fields.

These are component properties:

```
[(ngModel)]="user.age"
```

```
[(ngModel)]="firstName"
```

This will error out:

```
[(ngModel)]="fullName()"
```

What'd We Learn?

- The `[(ngModel)]` syntax allows us to specify a component property that will use two-way binding.
- Two-way binding means that if the component property is modified inside the component (JavaScript) or inside our web page (HTML), it will stay in sync.

Level 5

Services

Section 1

Revisiting Our Mock Data

car-parts.component.ts TypeScript

```typescript
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';

...
})
export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    this.carParts = CARPARTS;
  }
  ...
```

mocks.ts TypeScript

```typescript
import { CarPart } from './car-part';

export const CARPARTS: CarPart[] = [{
  "id": 1,
  "name": "Super Tires",
  "description": "These tires are the very best",
  "inStock": 5,
  "price": 4.99
}, { ... }, { ... }];
```

We are loading our `CarParts` by importing our mock file, but this isn't the best solution for working with data.

Why This Isn't the Best Method

- We'd need to import the mocks on every file that needs the data. If the way we access this data changes, we'd have to change it everywhere.
- It's not easy to switch between real and mock data.
- This sort of data loading is best left to service classes.

car-parts.component.ts TypeScript

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';

...
})
export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    this.carParts = CARPARTS;
  }
  ...
```

Introducing Services

Services are used to organize and share code across your app, and they're usually where we create our data access methods.

car-parts.component.ts

Asks the service for data

racing-data.service.ts

Fetches the appropriate data

mocks.ts

First, let's create the simplest service, and then we'll learn something called dependency injection to make it even more powerful.

Writing Our First Service

racing-data.service.ts TypeScript

```typescript
import { CARPARTS } from './mocks';

export class RacingDataService {
  getCarParts() {
    return CARPARTS;
  }
}
```

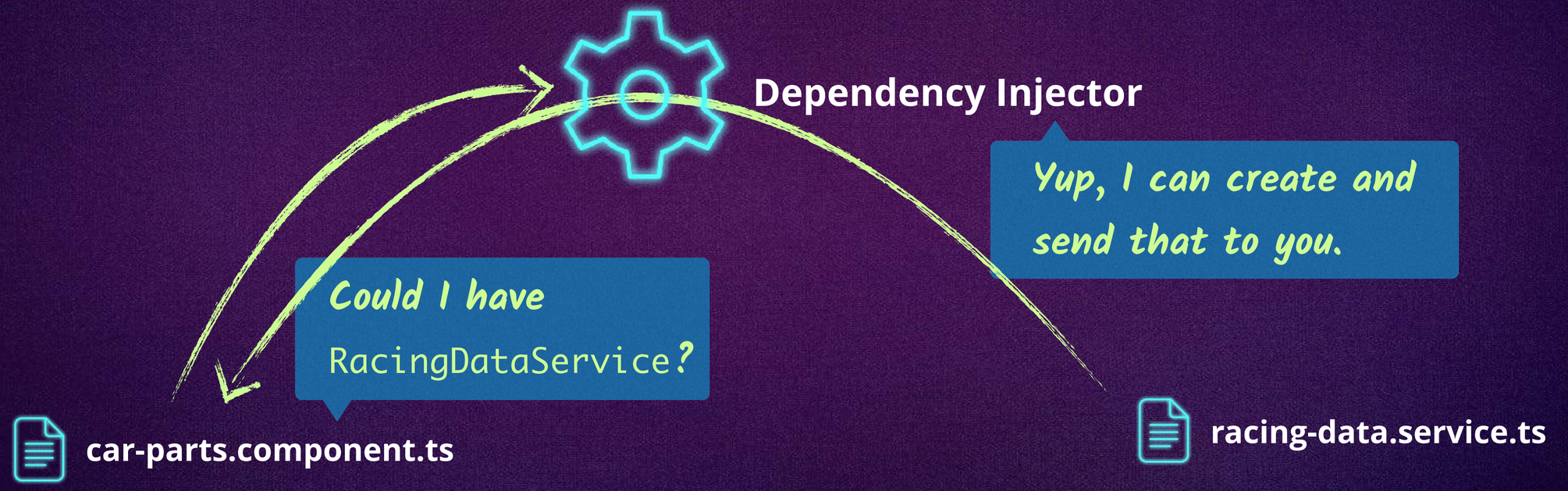
car-parts.component.ts TypeScript

```typescript
import { RacingDataService } from './racing-data.service';
...
export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    let racingDataService = new RacingDataService();
    this.carParts = racingDataService.getCarParts();
  }
}
```

- ✓ We've decoupled our data.
- ✗ Classes using this service must know how to create a `RacingDataService`.
- ✗ We'll be creating a new `RacingDataService` every time we need to fetch car parts.
- ✗ It'll be harder to switch between a mock service and a real one.

Introducing Dependency Injection

When you run an Angular 2 application, it creates a dependency injector. An injector is in charge of knowing how to create and send things.



The injector knows how to **inject** our **dependencies**.

Create (if needed) and send

Classes we depend on

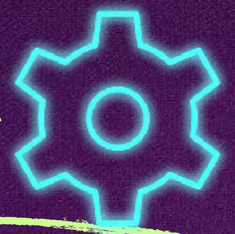
ACCELERATING THROUGH ANGULAR 2

Re-injecting Dependencies

If the injector already created a service, we can have it resend the same service.

Could I also have `RacingDataService`*?*

car-cart.component.ts



Dependency Injector

I already created one — here it is.

car-parts.component.ts

racing-data.service.ts

How Does an Injector Know What It Can Inject?

We must tell the injector what it can inject by registering "providers" with it.

These are the providers I have — they tell me what I can create and send.

Dependency Injector

There are three steps to make this all work with `RacingDataService`**:**

1. Add the injectable decorator to `RacingDataService`.
2. Let our injector know about our service by naming it as a provider.
3. Inject the dependency into our `car-parts.component.ts`.

Step 1: Add the Injectable Decorator

We need to turn our service into something that can be safely used by our dependency injector.

racing-data.service.ts TypeScript

```
import { CARPARTS } from './mocks';
import { Injectable } from '@angular/core';

@Injectable()
export class RacingDataService {
  getCarParts() {
    return CARPARTS;
  }
}
```

Don't forget the parentheses!

Step 2: Let Our Injector Know About the Service

We want all our subcomponents to have access to `RacingDataService`**. To do this, we register it as a provider at the top level — specifically,** AppModule**.**

main.ts TypeScript

```
...
import { RacingDataService } from './racing-data.service';

@NgModule({
  declarations: [ AppComponent ],
  imports: [ BrowserModule, FormsModule ]
  bootstrap: [ AppComponent ],
  providers: [ RacingDataService ]
})
class AppModule { }
...
```

Now all subcomponents can ask for (inject) our `RacingDataService` when they need it, and an instance of `RacingDataService` will either be delivered if it exists, or it will be created and delivered.

Edit: changed component provider to AppModule

Step 3: Injecting the Dependency

car-parts.component.ts TypeScript

```
...
import { RacingDataService } from './racing-data.service';

@Component({ ... })
export class CarPartsComponent {
  carParts: CarPart[];

  constructor(private racingDataService: RacingDataService) { }
}
```

Means TypeScript automatically defines component properties based on the parameters. The generated JavaScript is:

```
function CarPartsComponent(racingDataService) {
  this.racingDataService = racingDataService;
```

TypeScript syntax for setting the type of the parameter. This is what identifies that the `RacingDataService` should be injected into this component.

Using the Service to Fetch Our carParts

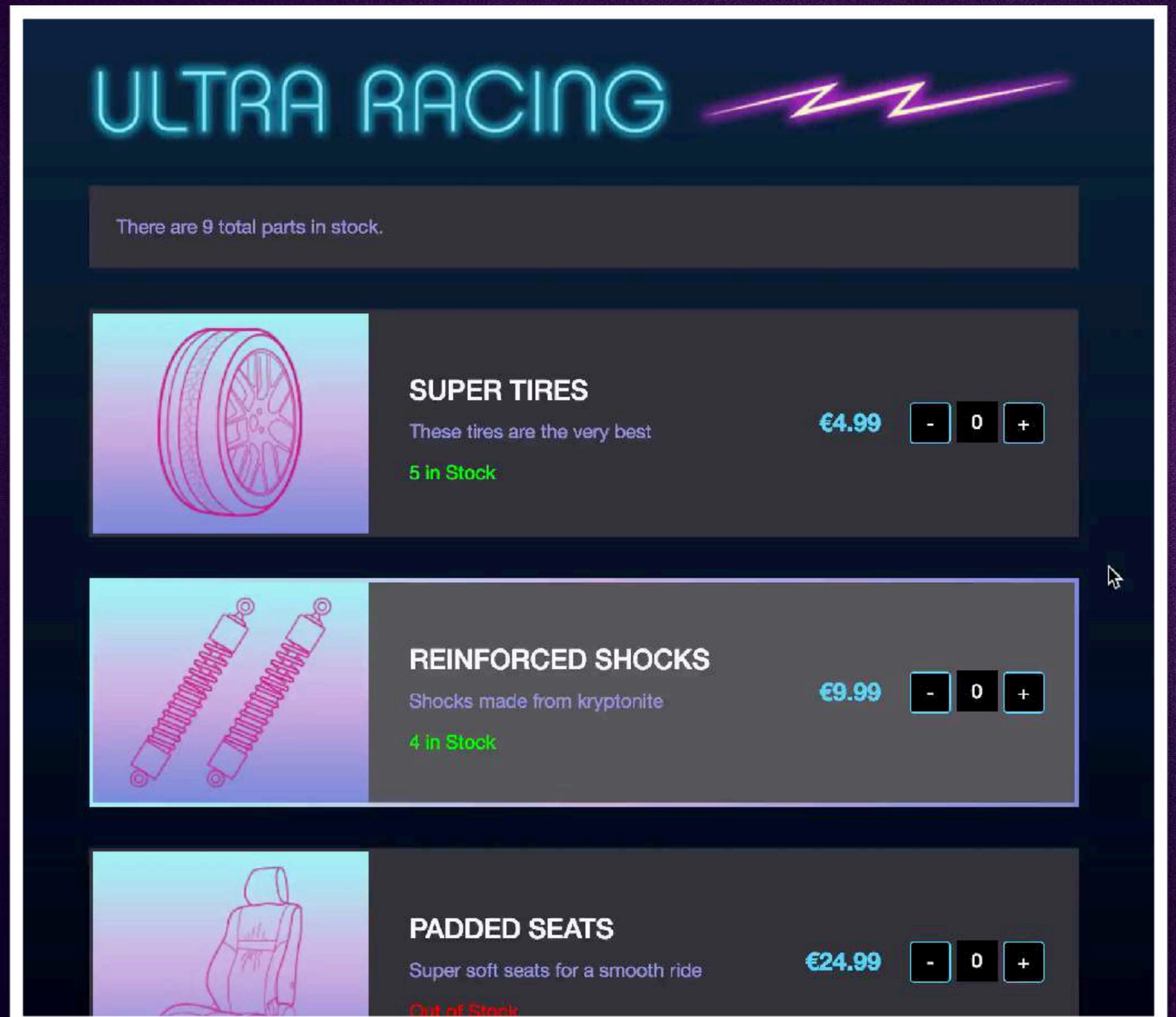
car-parts.component.ts TypeScript

```
...
import { RacingDataService } from './racing-data.service';

@Component({ ... })
export class CarPartsComponent {
  carParts: CarPart[];

  constructor(private racingDataService: RacingDataService) { }

  ngOnInit() {
    this.carParts = this.racingDataService.getCarParts();
  }
}
```

Now our app is more scalable and testable.

- **Scalable** because our dependencies aren't strongly tied to our classes.
- **Testable** because it'd be easy to mock services when we test the component.

Our App Still Works



What'd We Learn?

- Services are used to organize and share code across your app, and they're usually where you create your data access methods.
- We use dependency injection to create and send services to the classes that need them.
- We give our dependency injector providers so that it knows what classes it can create and send for us.
- We ask for dependencies by specifying them in our class constructor.

Level 5

Adding Http

Section 2

Let’s Use the Internet!

Up until now, we’ve been seeding our car parts with mock data. How might we fetch this from the internet instead?

car-parts.component.ts

Asks the service for data

racing-data.service.ts

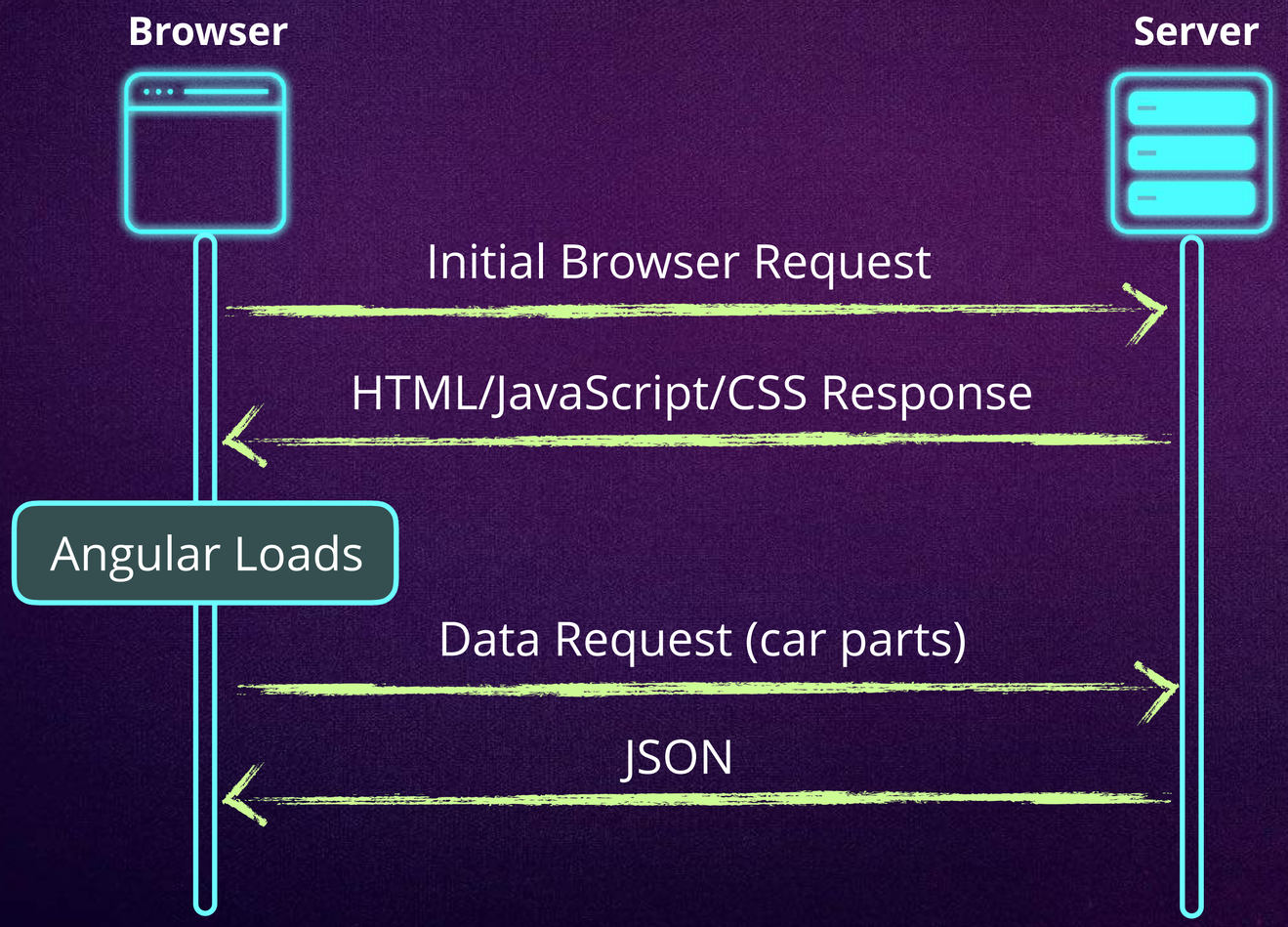
Fetches the appropriate data from the internet

Internet

This will be JSON data

Welcome to the Real World

When a user visits our web page, the Angular app loads first in our browser, and then it fetches the needed data.



Steps Needed to Use the HTTP Library

1. Create a JSON file with car parts data.
2. Ensure our application includes the libraries it needs to do `Http` calls.
3. Tell our injector about the `http` provider.
4. Inject the http dependency into our service and make the `http` `get` request.
5. Listen for data returned by this request.

car-parts.json

app.component.ts

racing-data.service.ts

car-parts.component.ts

Step 1: Creating a JSON File

We need to create a JSON data file to load from our service.

car-parts.json JSON

```
{
  "data": [
      {
        "id": 1,
        "name": "Super Tires",
        "description": "These tires are the very best",
        "inStock": 5,
        "price": 4.99,
        "image": "/images/tire.jpg",
        "featured": false,
        "quantity": 0
      },
      { ... },
      { ... }
  ]
}
```

We wrapped our array in an object to make it feel a little more realistic.

Step 2: Including the HTTP and RxJS Libraries

The HTTP library provides the get **call we will use to call across the internet.**

The RxJS library stands for Reactive Extensions and provides some advance tooling for our http **calls.**

If you used the 5-minute Quickstart, you have already included these libraries using **SystemJS**.

Step 3: Telling Our Injector It Can Inject HTTP

To inject the Http service we need to import the HttpModule.

main.ts TypeScript

```
...
import { HttpModule } from '@angular/http';

@NgModule({
  declarations: [ AppComponent ],
  imports: [ BrowserModule, FormsModule, HttpModule ],
  bootstrap: [ AppComponent ],
  providers: [ RacingDataService ],
})
class AppModule { }
...
```

@angular/http TypeScript

Http is now available as a provider

```
@NgModule({
  providers: [ Http, ... ],
})
export class HttpModule { }
...
```

Now we'll be able to inject the HTTP library when we need it.

Step 4: Injecting HTTP & Using It

Now let's inject http **and call out to the internet.**

racing-data.service.ts TypeScript

```typescript
import { Injectable } from '@angular/core';
import { CarPart } from './car-part';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class RacingDataService {

  constructor(private http: Http) { }

  getCarParts() {
    return this.http.get('app/car-parts.json')
                .map(response => <CarPart[]>response.json().data);
  }
}
```

Injecting HTTP as a dependency

We can do this because our service class is injectable

There's a lot going on here — let's break it out.

Stepping Through Our get Request

You might expect `get` to return a promise, but this returns an observable. Observables give us additional functionality on our http calls — one of which is to treat the return value like an array.

```
getCarParts() {
  return this.http.get('app/car-parts.json')
  .map(response =>                <CarPart[]>                response.json()                .data);
```

For the data returned, do this to the response.

Tells our TypeScript compiler to treat this like an array of `CarParts`.

For each response, parse the string into JSON.

The array we want is under the `data` keyword.

Step 5: Subscribing to the Stream

Since our service now returns an observable object, we need to subscribe to that data stream and tell our component what to do when our data arrives.

car-parts.component.ts TypeScript

```
...
export class CarPartsComponent {

  constructor(private racingDataService: RacingDataService) { }

  ngOnInit() {
    this.racingDataService.getCarParts()
        .subscribe(carParts => this.carParts = carParts);
  }
  ...
```

When `carParts` arrive on our data stream, set it equal to our local `carParts` array.

Solving the Problem

In our browser, we get nothing:

```
TypeError: Cannot read property 'length' of undefined at
CarPartsComponent.totalCarParts
```

car-parts.component.ts TypeScript

```typescript
totalCarParts() {
  let sum = 0;

  for (let carPart of this.carParts) {
     sum += carPart.inStock;
  }

  return sum;
}
```

Not an array yet

When our page initially loads, we haven't yet fetched any data, so our `carParts` is **undefined**.

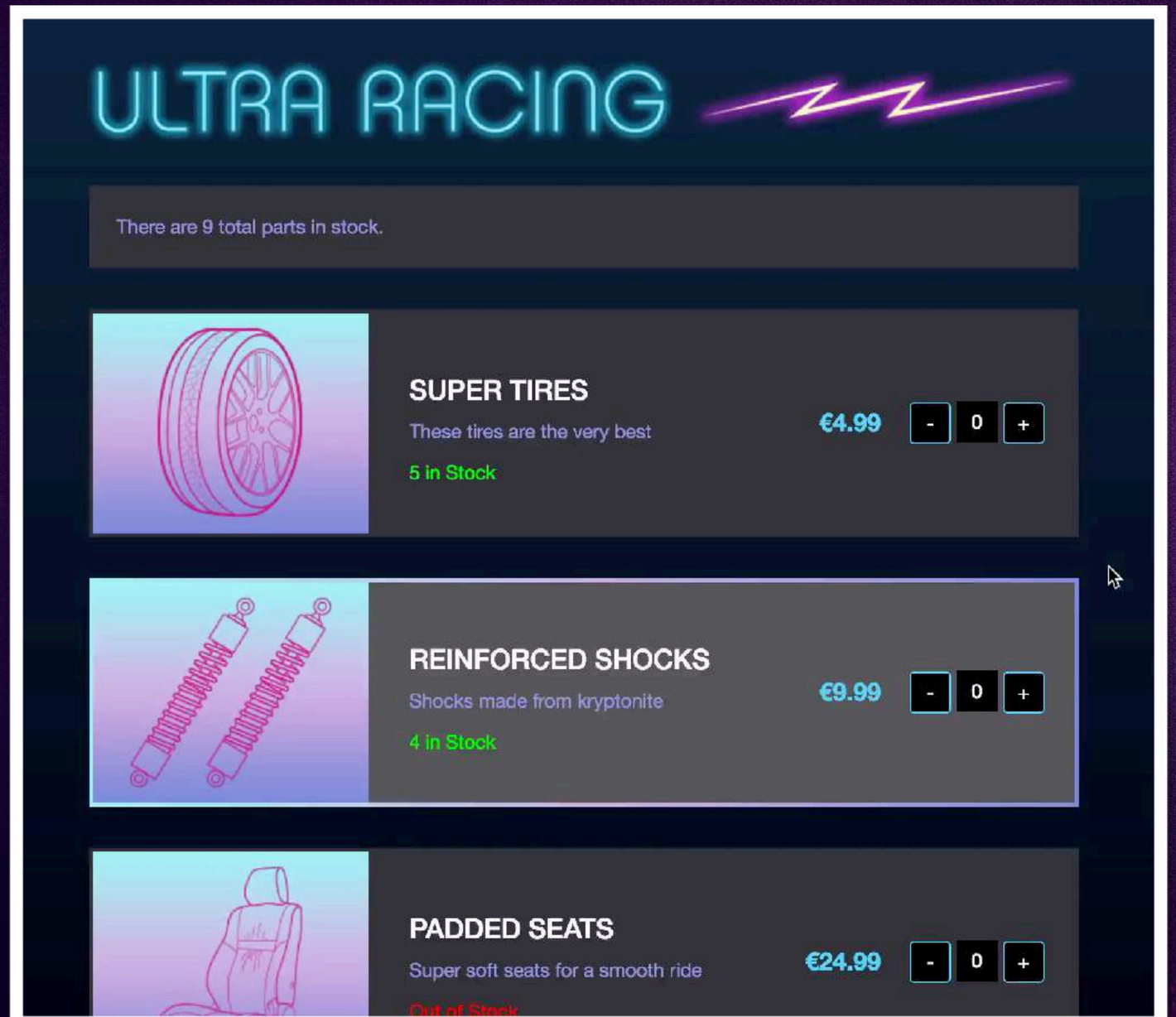
Checking for Undefined carParts

Let's make sure we only calculate sum if carParts **is an array.**

car-parts.component.ts TypeScript

```typescript
totalCarParts() {
  let sum = 0;

  if (Array.isArray(this.carParts)) {
    for (let carPart of this.carParts) {
      sum += carPart.inStock;
    }
  }
  return sum;
}
```

All Working — Now Over the Internet!



Last Thoughts on Implementation

1. We didn’t do any error handling. If you're writing a production app, you’d want to do this.

2. Since we isolated our network calls as a service, we could easily write a `RacingDataServiceMock` service and inject it when we’re testing or developing offline.

3. Observables can be quite powerful and are worth learning about if you are making lots of `http` calls.

What'd We Learn?

- Angular apps usually load data using service classes after the Angular app is initialized and running.
- We can use the HTTP library through dependency injection.
- Our http calls return an observable, not a promise, which behaves more like an array.