Desarrollo AppWeb Multip R.Oliva 07.2020

Apuntes propios + ejercicios / Docente Brian Ducca - Upd. 29.7.2020

PARTE I / Usando Freeman
0) Material:

a) Apunte de B.Ducca en: C:\MIoT2020\Materias\DesarrolloAppMultiplat\delGitHub\Material\Material DAM.pdf
b) Repo: https://github.com/brianducca/dam
c) Books: c.1) ProAngular9 – Freeman / Springer (carece de component treatment), que a su vez tiene su propio repo:
   https://github.com/Apress/pro-angular-9
   c.2) LEARNING ANGULAR, STEP BY STEP Angular Up & Running - Shyam Seshadri O'Reilly 2018 (este ultimo tiene components) → para hacer las prácticas de clase.
   Repo libro: https://github.com/shyamseshadri/angular-up-and-running

1a. Primera clase 24.6.20 – Instalación de Node 12 ok y Framework Angular en Probook / Ubuntu 18.04 ok

```
 node -v  → v12.18.1
 npm -v   →v6.14.5
npm install –g @angular /cli
```

Se trabaja desde VSC, agregamos componente Typescript Importer

1b. SPA – todas las pantallas en una misma página. Evita requests al servidor para cargar mas HTML. Se trabaja con el Framework Angular y específicamente con Ionic para el FrontEnd (que permite Apps móviles o Web) y con Node para el Backend. La conexión a DB puede ser via mySQL o una noSQL como Mongo.

1c. Estructura: basado en módulos (clases Typescript), que contienen componentes (cada uno con Vista, Estilo, Controlador (Typescript) y PruebaUnitaria. Hay un modulo raíz App Module que es la base, la estructura es la de un árbol jerárquico.

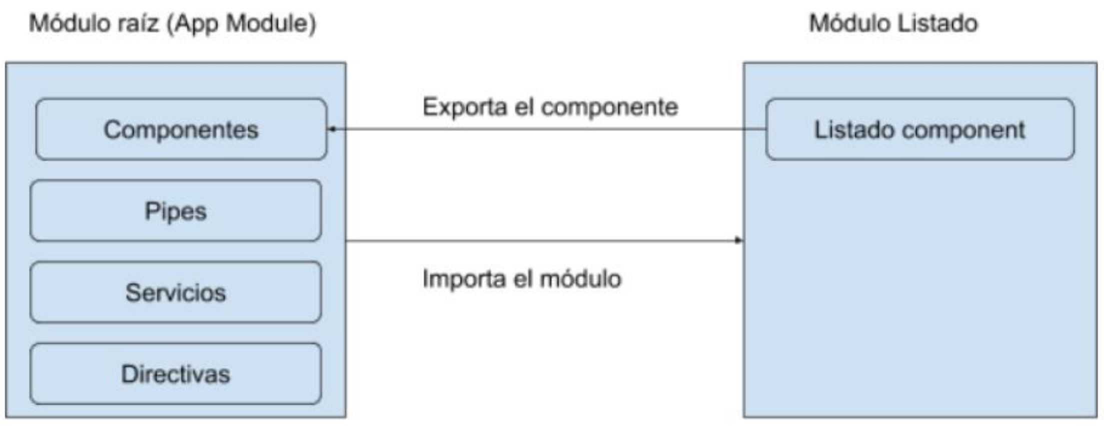


Figura 1 – App Module principal y module Listado

1d. Bindings → entre el DOM y los componentes se establecen distintos tipos de Bindings:

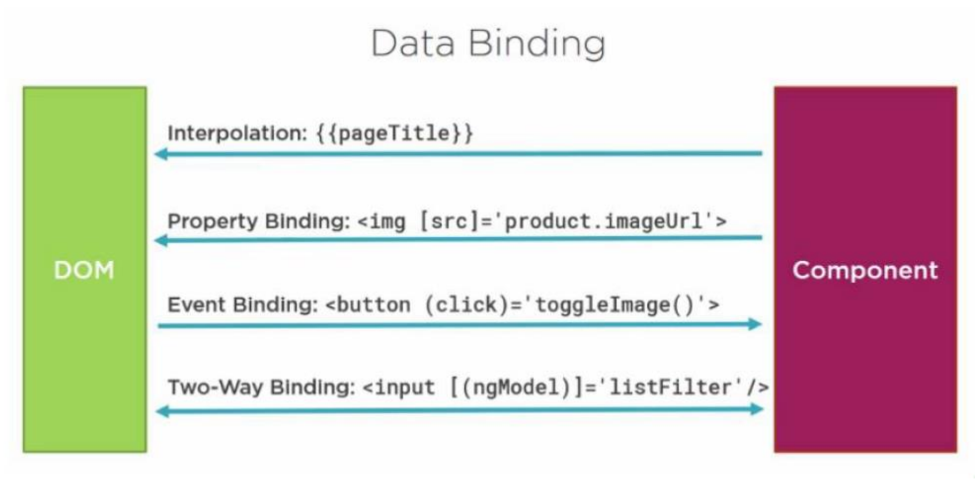

Data Binding
DOM
Component
Interpolation: {{pageTitle}}
Property Binding: <img [src]='product.imageUrl'>
Event Binding: <button (click)='toggleImage()'>
Two-Way Binding: <input [(ngModel)]='listFilter'/>

Figura 2 – Bindings

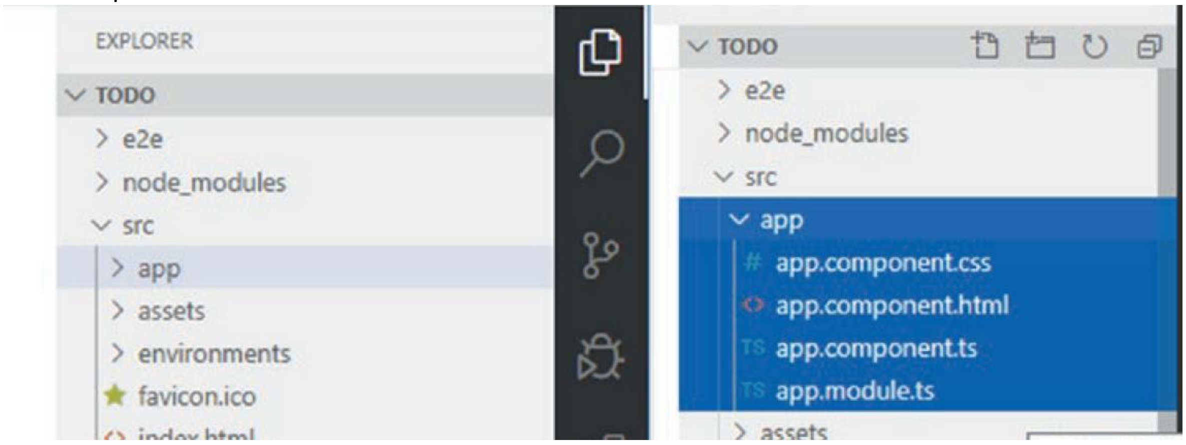
Ejemplos de SPAs: Netflix, gMail, github, etc.
Hay otros frameworks similares como React y Vue.

2. Ejemplo "todo" (do-list elemental) del libro de Freeman. Primero crea el Project todo con:

Listing 2-5. Creating the Angular Project

```
ng new todo --routing false --style css --skip-git --skip-tests
```

Debería quedar esto:



2.1.a) Luego se compila con:

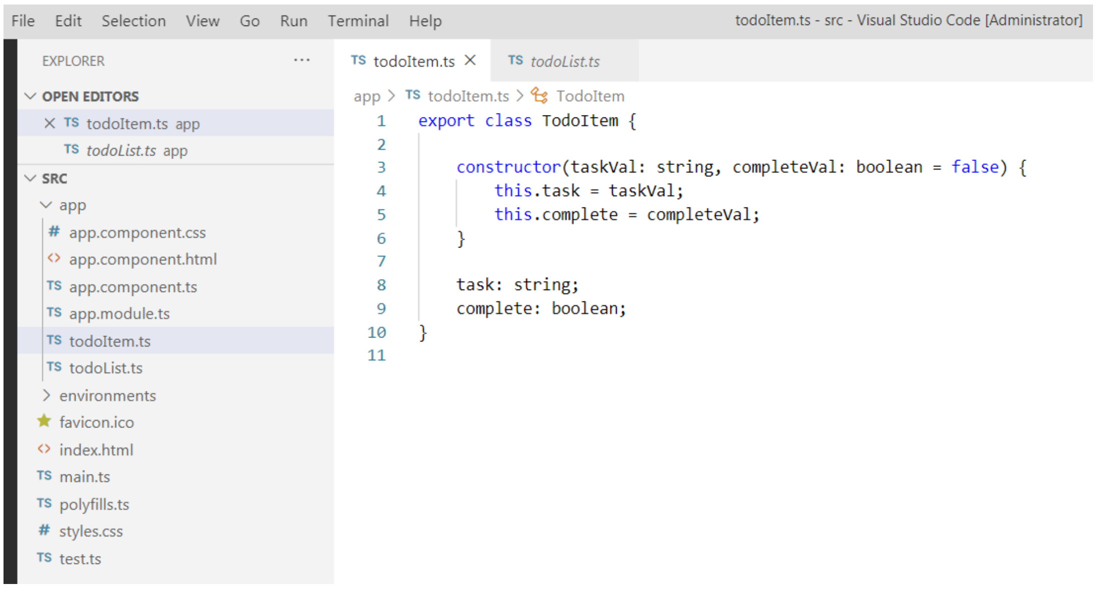
Listing 2-6. Starting the Angular Development Tools

```
ng serve
```

Y hay que esperar un rato, pero si todo va bien:

```
** Angular Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **
: Compiled successfully.

Date: 2020-02-09T11:23:47.497Z - Hash: 3f280025364478cce5b4
5 unchanged chunks

Time: 465ms
: Compiled successfully.
```

2.2) Arranca definiendo el Data Model, a partir del archivo `todoItem.ts`, que se coloca en /src



```
export class TodoItem {

    constructor(taskVal: string, completeVal: boolean = false) {
        this.task = taskVal;
        this.complete = completeVal;
    }

    task: string;
    complete: boolean;
}
```

<u>Notas del Autor:</u>

- `todoitem.ts` defines TodoItem class:

*a) The **export, class, and constructor** keywords are standard JavaScript. Not all browsers support these features, which are relatively recent additions to the JavaScript specification, and the build process for Angular applications can translate this type of feature into code that older browsers can understand (Chapter 11)*

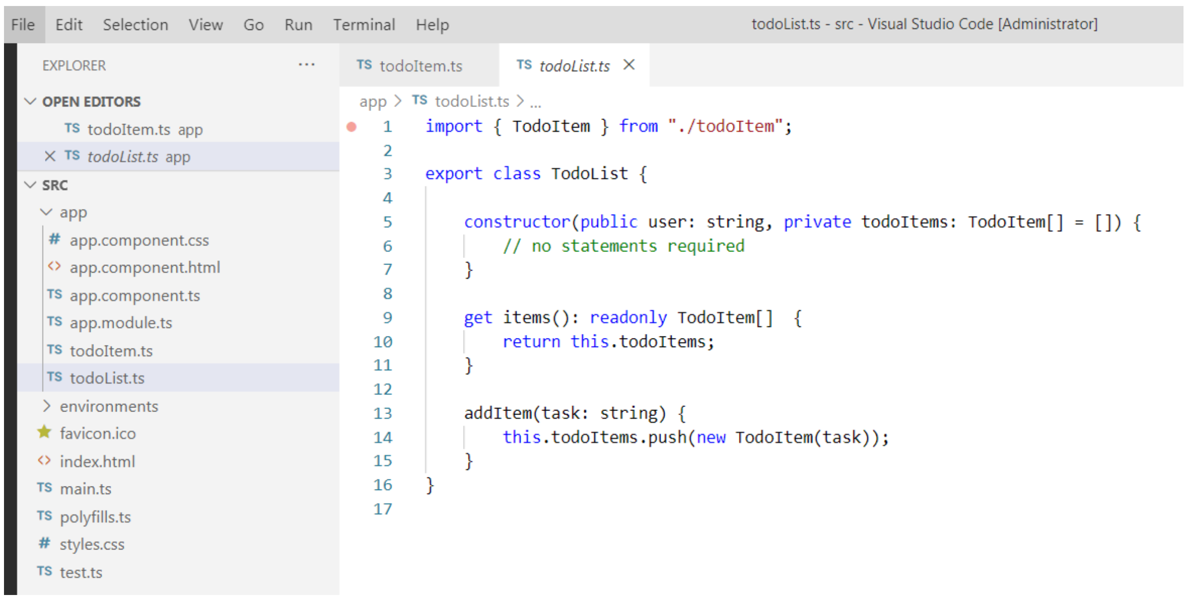
*b) The **export** keyword relates to JavaScript modules. When using modules, each TypeScript or JavaScript file is considered to be a self-contained unit of functionality, and the export keyword is used to identify data or types that you want to use elsewhere in the application. JavaScript modules are used to manage the dependencies that arise between files in a project and avoid having to manually manage a complex set of script elements in the HTML file (Chapter 11 for details of how modules work).*
*c) The **class** keyword declares a class, and*
*d) the **constructor** keyword denotes a class constructor. Unlike other languages, such as C#, JavaScript doesn't use the name of the class to denote the constructor.*
e) We use TS's concise constructor:

```
        constructor(taskVal: string, completeVal: boolean = false)
```

2.3) Arranca definición de lista de *todos*:

*To create a class that represents a list of to-do items, I added a file named **todoList.ts** to the src/app folder.*

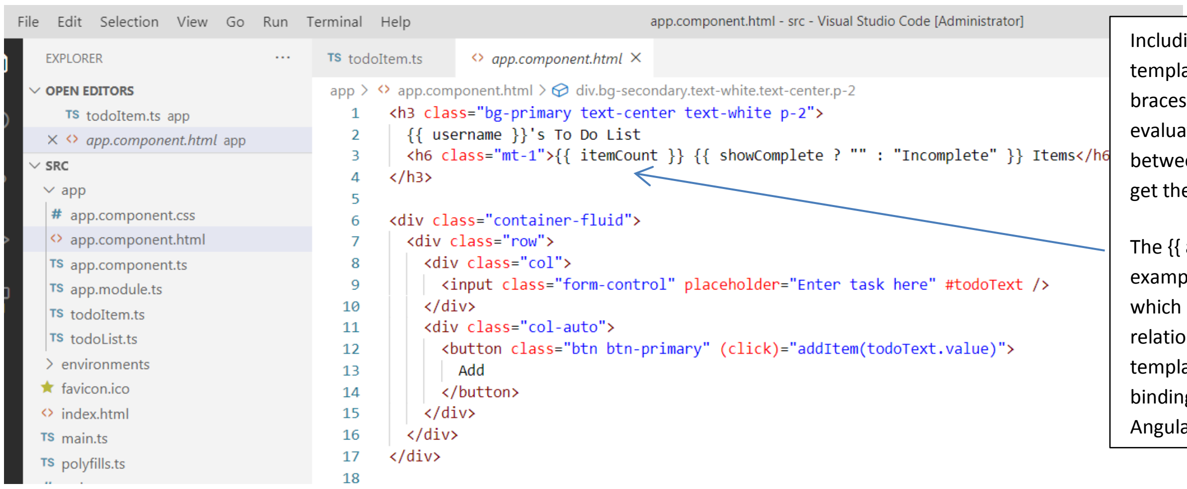
- *The import keyword declares a dependency on the TodoItem class and specifies the code file that defines it.*
- *The TodoList class defines a constructor that receives the initial set of to-do items. I don't want to give unrestricted access to the array of TodoItem objects, so I have defined a property named items that returns a read-only array, which is done using the readonly keyword.*



```
import { TodoItem } from "./todoItem";

export class TodoList {

    constructor(public user: string, private todoItems: TodoItem[] = []) {
        // no statements required
    }

    get items(): readonly TodoItem[]  {
        return this.todoItems;
    }

    addItem(task: string) {
        this.todoItems.push(new TodoItem(task));
    }
}
```

2.3) Se busca una forma de mostrar la lista en el DOM:

*I need a way to display the data values in the model to the user. In Angular, this is done using a template, which is a fragment of HTML that contains expressions that are evaluated by Angular and that inserts the results into the content that is sent to the browser. The **angular-cli** setup for the project created a template file called **app.component.html** in the **src/app folder**. I edited this file to remove the placeholder content and add the content shown in Listing 2-9:*



```
<h3 class="bg-primary text-center text-white p-2">
  {{ username }}'s To Do List
  <h6 class="mt-1">{{ itemCount }} {{ showComplete ? "" : "Incomplete" }} Items</h6
</h3>

<div class="container-fluid">
  <div class="row">
    <div class="col">
      <input class="form-control" placeholder="Enter task here" #todoText />
    </div>
    <div class="col-auto">
      <button class="btn btn-primary" (click)="addItem(todoText.value)">
        Add
      </button>
    </div>
  </div>
</div>
```

Including a data value in a template is done using double braces—{{ and }}—and Angular evaluates whatever you put between the double braces to get the value to display.

The {{ and }} characters are an example of a **data binding**, which means they create a relationship between the template and a data value. Data bindings are an important Angular feature

2.4) Apenas se guarda el archivo, Angular intenta compilarlo y da error. Esto es porque está aun incompleto el app.component.ts, Se *debe agregar*

2.4.1) referencias a los archivos recién creados (lo primero que aparece):

...

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

import { TodoList } from "./todoList";

import { TodoItem } from "./todoItem";

...

Listing 2-10. Editing the Contents of the app.component.ts File in the src/app Folder

```
import { Component } from '@angular/core';
import { TodoList } from "./todoList";
import { TodoItem } from "./todoItem";

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  private list = new TodoList("Bob", [
    new TodoItem("Go for run", true),
    new TodoItem("Get flowers"),
    new TodoItem("Collect tickets"),
  ]);

  get username(): string {
    return this.list.user;
  }

  get itemCount(): number {
    return this.list.items
      .filter(item => !item.complete).length;
  }
}
```

This is an example of a decorator, which provides metadata about a class. This is the *@Component* decorator, and, as its name suggests, it tells Angular that this is a component. The decorator provides configuration information through its properties. This @Component decorator specifies three properties:

- `selector`,
- `templateUrl`
- `styleUrls`.

This is the AppComponent class, which Angular intantiates to create the component, which has:

- a private list property, which is assigned a TodoList object, and is populated with an array of TodoItem objects. –

- also read-only properties named
    `username`
    `itemCount`
that rely on the TodoList object to produce their values. The username property returns the value of the TodoList.user property, and the itemCount -> explained in TEXT →

Notas sobre 2.4.1)

IMPORTs:

- *The first import statement is used in the listing to load the @angular/core module, which contains the key Angular functionality, including support for components. When working with modules, the import statement specifies the types that are imported between curly braces. In this case, the import statement is used to load the Component type from the module. The @angular/core module contains many classes that have been packaged together so that the browser can load them all in a single JavaScript file.*
- *The other import statements are used to declare dependencies on the data model classes defined earlier. The target for this kind of import starts with ./, which indicates that the module is defined relative to the current file.*

@Component:

- *1.selector: The selector property specifies a CSS selector that matches the HTML element to which the component will be applied. The **app-root** element specified by this decorator is the default set by the angular-cli package. It corresponds to an HTML element that was added to the **index.html** file, which you can find in the src folder and that was created with the following content:*

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Todo</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

2.template-url: The property is to specify the component's template, which is the app.component.html file for this component.

3.styleUrls: The styleUrls property specifies one or more CSS stylesheets that are used to style the elements produced by the component and its template.

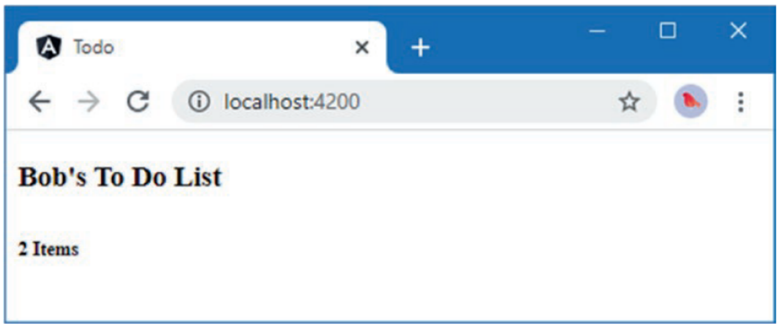
Note on getItemCount():
*The value for the itemCount property is produced using a **lambda function**, also known as a **fat arrow** function, which is a more concise way of expressing a standard JavaScript function.*

```
get itemCount(): number {
  return this.list.items.filter(item => !item.complete).length;
}
```

The arrow in the lambda expressions is read as "goes to" such as "item goes to not item.complete." Lambda expressions are a recent addition to the JavaScript language specification (std within Typescript), and they provide an alternative to the conventional JS way of using functions as arguments like this:

```
...
return this.model.items.filter(function (item) { return !item.complete });
...
```

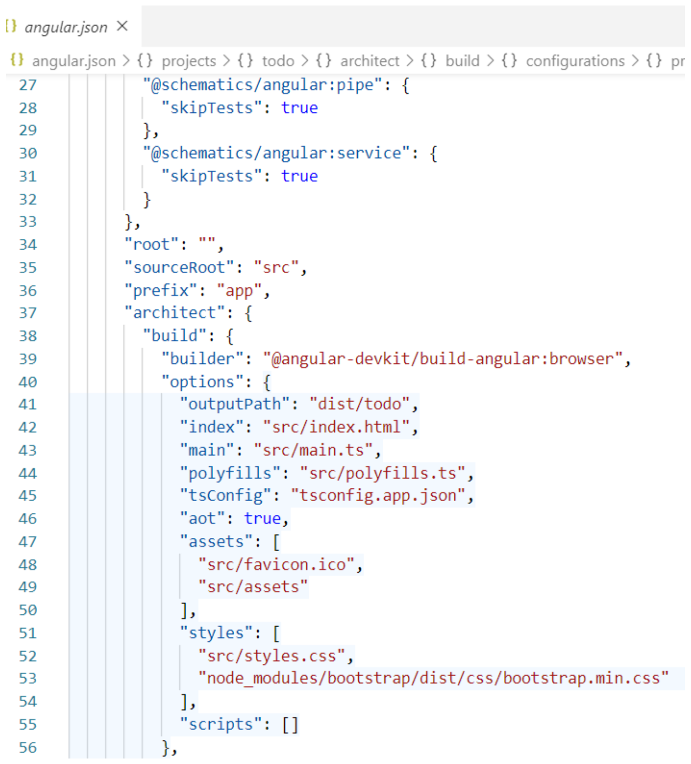
Salida de 2.4) modificado



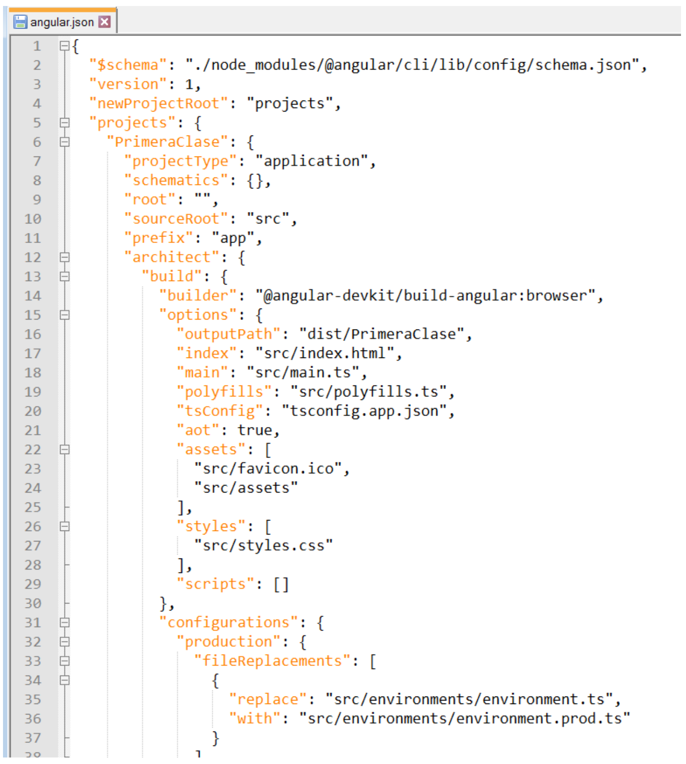
Bob's To Do List

2 Items

2.5) Agregamos Bootstrap CSS
Styling theHTML Elements: para mejorar el aspecto, en el ejemplo se utiliza bootstrapCSS
Esto se hace en el angular.json, en este caso es el de la izq. que en "styles" agrega `"src/styles.css","node_modules/bootstrap/dist/css/bootstrap.min.css"` *, en el de la derecha vemos el de PrimeraClase con Brian, que en la parte de "styles" sólo usa src/styles.css:*


angular.json
angular.json > {} projects > {} todo > {} architect > {} build > {} configurations > {} pr

```
        "@schematics/angular:pipe": {
          "skipTests": true
        },
        "@schematics/angular:service": {
          "skipTests": true
        }
      },
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/todo",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": true,
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "src/styles.css",
              "node_modules/bootstrap/dist/css/bootstrap.min.css"
            ],
            "scripts": []
          },
```



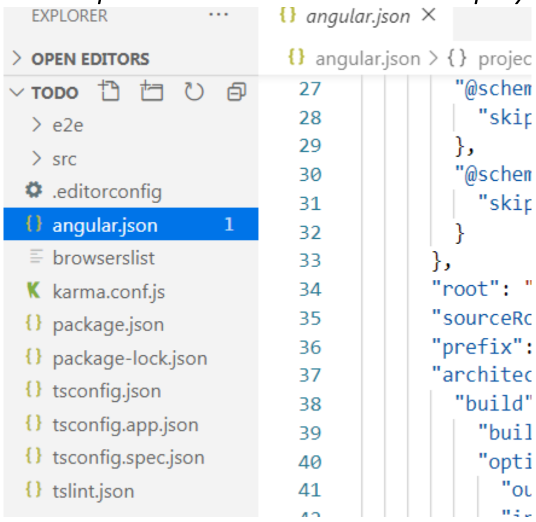
```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "PrimeraClase": {
      "projectType": "application",
      "schematics": {},
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/PrimeraClase",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": true,
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "src/styles.css"
            ],
            "scripts": []
          },
          "configurations": {
            "production": {
              "fileReplacements": [
                {
                  "replace": "src/environments/environment.ts",
                  "with": "src/environments/environment.prod.ts"
                }
```

El paquete bootstrap lo agregan con:

Listing 2-11. Adding a Package to the Example Project

```
npm install bootstrap@4.4.1
```

Para modificar el angular.json, recordar que está en el directorio raíz del proyecto, en el caso del libro es /todo:



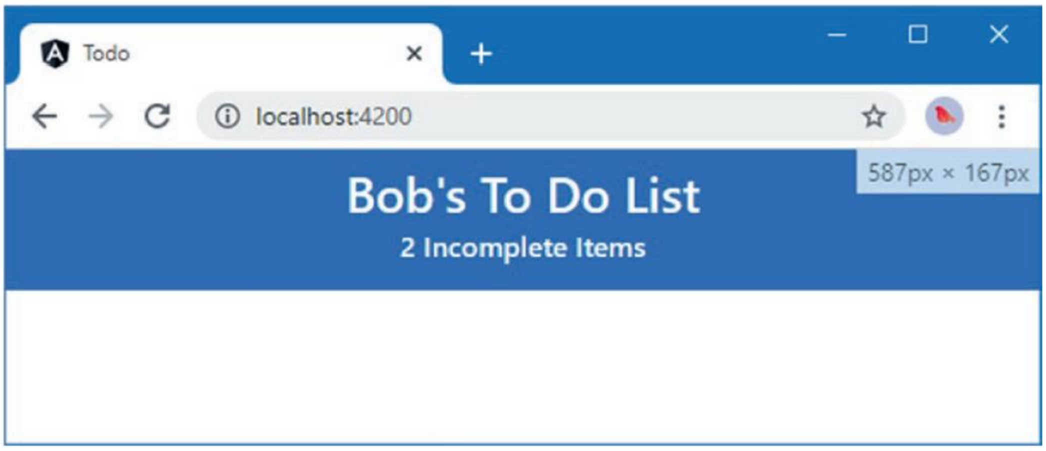
```
          "@schem
            "skip
          },
          "@schem
            "skip
          }
        },
        "root": "
        "sourceRc
        "prefix":
        "architec
          "build"
            "buil
            "opti
              "ou
```

the angular.json file is used to configure the project tools, and the statement shown in the listing incorporates the Bootstrap CSS file into the project so that it will be included in the content sent to the browser. Bootstrap works by adding elements to classes. In Listing 2-13, I have added the elements in the template to classes that will change their appearance.

Listing 2-13. Styling Content in the app.component.html File in the src/app Folder

```
<h3 class="bg-primary text-center text-white p-2">
  {{ username }}'s To Do List
  <h6 class="mt-1">{{ itemCount }} Incomplete Items</h6>
</h3>
```

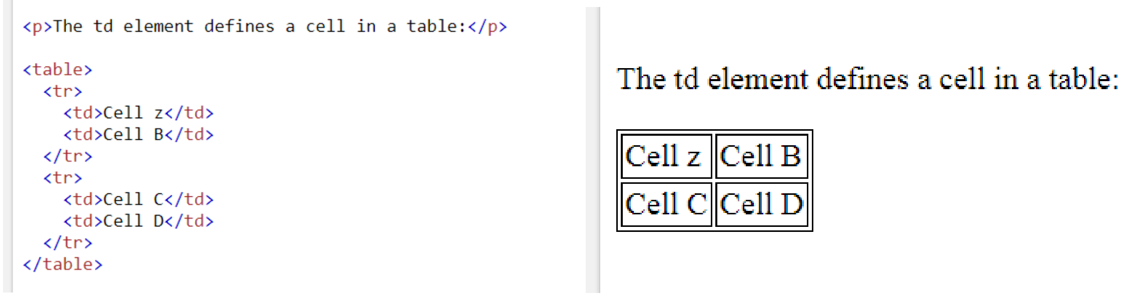
Salida de 2.5) modificado



Bob's To Do List

2 Incomplete Items

REPASO Inserción de elementos de una tabla en HTML:
(https://www.w3schools.com/tags/tryit.asp?filename=tryhtml_td)

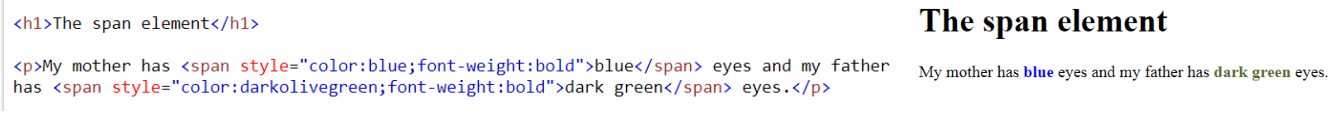


```
<p>The td element defines a cell in a table:</p>

<table>
  <tr>
    <td>Cell z</td>
    <td>Cell B</td>
  </tr>
  <tr>
    <td>Cell C</td>
    <td>Cell D</td>
  </tr>
</table>
```

The td element defines a cell in a table:

Cell z	Cell B
Cell C	Cell D

Y el otro elemento que se usará es <span>, </span>: "a) The <span> tag is an inline container used to mark up a part of a text, or a part of a document. b) The <span> tag is easily styled by CSS or manipulated with JavaScript using the class or id attribute. C) The <span> tag is much like the <div> element, but <div> is a block-level element and <span> is an inline element.

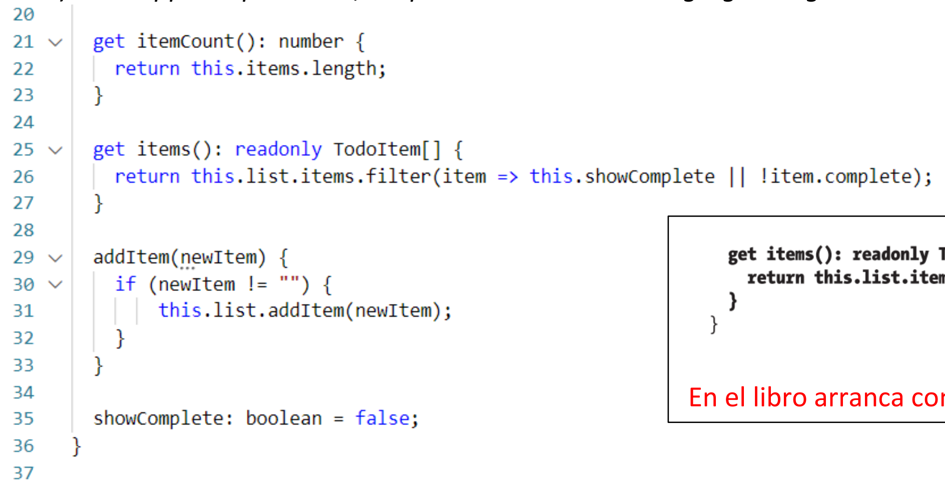


```html
<h1>The span element</h1>

<p>My mother has <span style="color:blue;font-weight:bold">blue</span> eyes and my father
has <span style="color:darkolivegreen;font-weight:bold">dark green</span> eyes.</p>
```

The span element

My mother has **blue** eyes and my father has **dark green** eyes.

2.5.1) En el app.component.ts, después de itemcount se agrega lo siguiente:



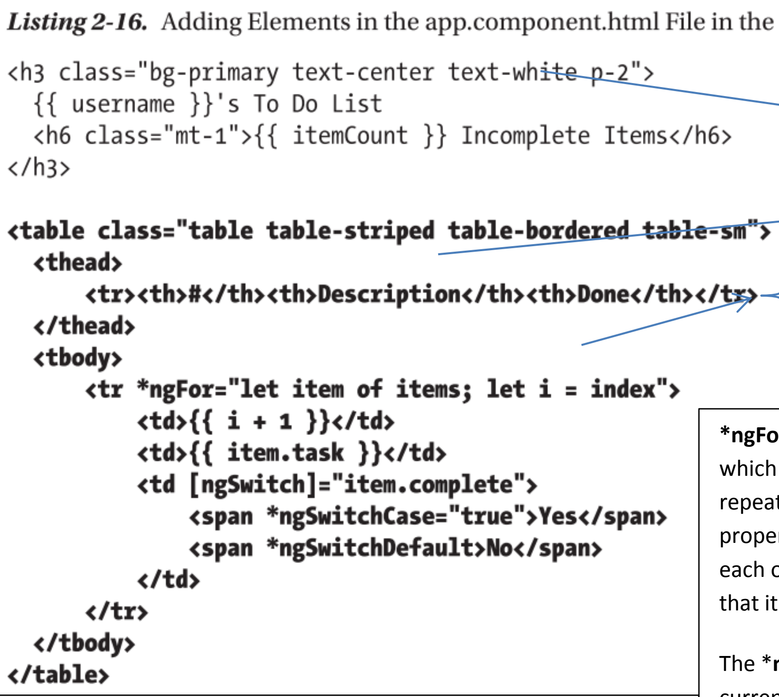
```typescript

  get itemCount(): number {
    return this.items.length;
  }

  get items(): readonly TodoItem[] {
    return this.list.items.filter(item => this.showComplete || !item.complete);
  }

  addItem(newItem) {
    if (newItem != "") {
      this.list.addItem(newItem);
    }
  }

  showComplete: boolean = false;
}

```

```typescript
    get items(): readonly TodoItem[] {
      return this.list.items;
    }
  }
```

En el libro arranca con este.. (B)

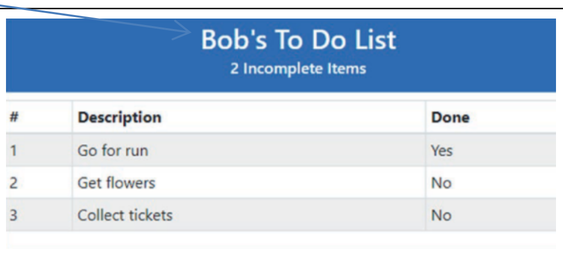
Para mostrarlo en el DOM se utiliza el agregado en app.component.html, recordando que <tr></tr> define un "table row" y <td></td> una celda del row en que estemos. A su vez, <th></th> indica el header de la tabla.

Javascript expressions directly in bindings like this, but for



Listing 2-16. Adding Elements in the app.component.html File in the src/app Folder

```html
<h3 class="bg-primary text-center text-white p-2">
  {{ username }}'s To Do List
  <h6 class="mt-1">{{ itemCount }} Incomplete Items</h6>
</h3>

<table class="table table-striped table-bordered table-sm">
  <thead>
      <tr><th>#</th><th>Description</th><th>Done</th></tr>
  </thead>
  <tbody>
      <tr *ngFor="let item of items; let i = index">
          <td>{{ i + 1 }}</td>
          <td>{{ item.task }}</td>
          <td [ngSwitch]="item.complete">
              <span *ngSwitchCase="true">Yes</span>
              <span *ngSwitchDefault>No</span>
          </td>
      </tr>
  </tbody>
</table>
```



Bob's To Do List

2 Incomplete Items

#	Description	Done
1	Go for run	Yes
2	Get flowers	No
3	Collect tickets	No

ngFor:** This expression tells Angular to treat the ***tr element to which it has been applied as a template that should be repeated for every object returned by the component's ***items*** property. The ***let item*** part of the expression specifies that each object should be assigned to a variable called ***item*** so that it can be referred to within the template.

The ***ngFor** expression also keeps track of the **index** of the current object in the array that is being processed, and this is assigned to a second variable called ***i***.

{{ i + 1}} :For simple transformations, you can embed your Javascript expressions directly in bindings like this, but for more complex operations, angular has a feature called ***pipes***

ngSwitch:** The [ngSwitch] expression is a conditional statement that is used to insert different sets of elements into the document based on a specified value, which is the ***item.complete property in this case. Nested within the ***td*** element are two span elements that have been annotated with *ngSwitchCase and ***ngSwitchDefault and that are equivalent to the case and default keywords of a regular JavaScript switch block** . the result is that the first span element is added to the document when the value of the ***item.complete*** property is true, and the second span element is added to the document when ***item.complete*** is false. The result is that the true/false value of the ***item.complete*** property is transformed into span elements ***containing either Yes or No***

18.07.20

2.6) Agregamos TwoWay Data Binding: At the moment, the template contains only one-way data bindings, which means they are used to display a data value but are unable to change it. Angular also supports two-way data bindings, which can be used to display a data value and change it, too. → Checkbox

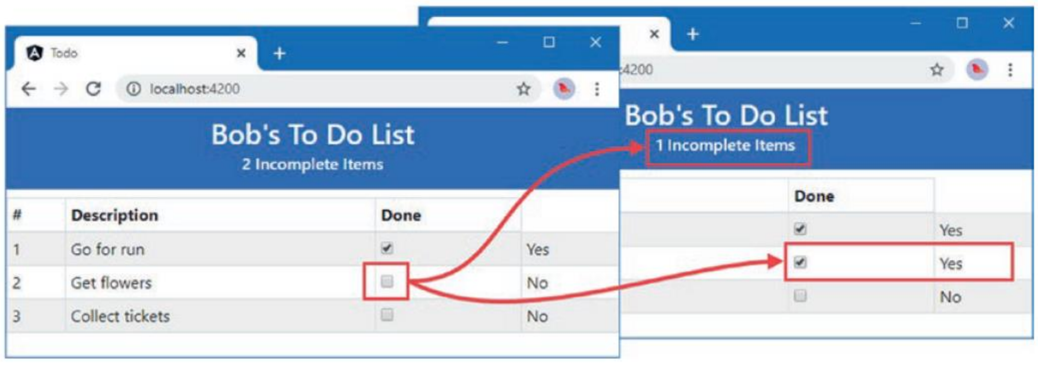
```
<tbody>
    <tr *ngFor="let item of items; let i = index">
        <td>{{ i + 1 }}</td>
        <td>{{ item.task }}</td>
        <td><input type="checkbox" [(ngModel)]="item.complete" /></td>
        <td [ngSwitch]="item.complete">
            <span *ngSwitchCase="true">Yes</span>
            <span *ngSwitchDefault>No</span>
        </td>
```

The ***ngModel*** template expression creates a two-way binding between a data value (the ***item.complete*** property in this case) and a form element, in this case an input element. When you save the changes to the template, you will see a new column that contains checkboxes appear in the table. The initial value of the checkbox is set using the ***item.complete*** property, just like a regular one-way binding, but when the user toggles the checkbox, Angular responds by updating the specified model property.
When you save the changes to the template, the Angular development tools will report an error because the ***ngModel*** feature has not been enabled. Angular applications have a root module, which is used to configure the application. The root module for the example application is defined in the ***app.module.ts*** file:

Listing 2-18. Enabling a Feature in the app.module.ts File in the src/app Folder

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from "@angular/forms";

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

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

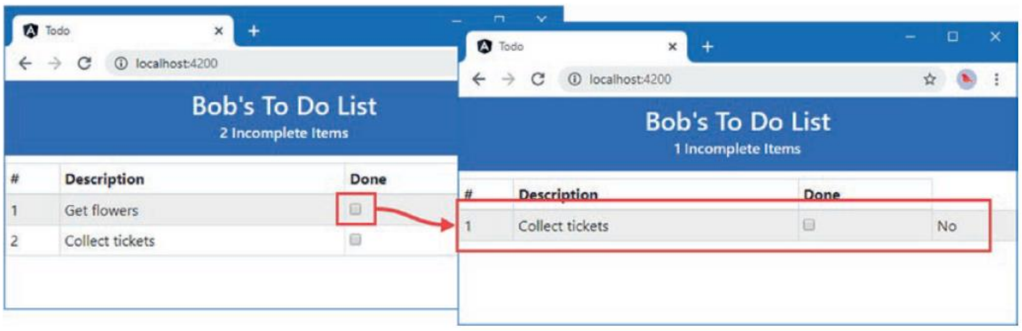
The functionality provided by Angular is shipped in separate JavaScript modules, which must be added to the application with an import statement and registered using the imports property defined by the ***NgModule*** decorator. When the Angular development tools build the application, they incorporate the features specified by the imports property into the files that are sent to the browser. (save and run ***ng serve*** again, to see changes).



(when checkbox ticked, the Nº of incomplete items is also decreased)

2.7) Agregamos Filtrado: The checkboxes allow the data model to be updated, and the next step is to remove to-do items once they have been marked as done. Now we change the component's items property so that it filters out any items that have been completed. Since the data model is *live* and changes are reflected in data bindings immediately, checking the checkbox for an item removes it from view This is done in the ***app.component.ts*** File in the src/app Folder (tal cual se aclaró en 2.5.1):

```
  get username(): string {
    return this.list.user;
  }

  get itemCount(): number {
    return this.items.length;
  }

  get items(): readonly TodoItem[] {
    return this.list.items.filter(item => !item.complete);
  }
}
```



Bob's To Do List
2 Incomplete Items

#	Description	Done
1	Get flowers	☐
2	Collect tickets	☐

Bob's To Do List
1 Incomplete Items

#	Description	Done	
1	Collect tickets	☐	No

2.8) Agregamos Nuevos ítems: This is done in the template after the Incomplete Items counter, it adds elements to the template that will allow the user to enter details of a task:

```
<h3 class="bg-primary text-center text-white p-2">
  {{ username }}'s To Do List
  <h6 class="mt-1">{{ itemCount }} Incomplete Items</h6>
</h3>

<div class="container-fluid">
  <div class="row">
    <div class="col">
      <input class="form-control" placeholder="Enter task here" #todoText />
    </div>
    <div class="col-auto">
      <button class="btn btn-primary" (click)="addItem(todoText.value)">
        Add
      </button>
    </div>
  </div>
</div>

<div class="m-2">
  <table class="table table-striped table-bordered table-sm">
    <thead>
        <tr><th>#</th><th>Description</th><th>Done</th></tr>
```

Most of the new elements create a grid layout to display an ***input element*** and a ***button element***. The ***input element*** has an attribute whose name starts with the # character (#todoText), which is used to define a variable (#todoText), to refer to the element in the template's data bindings.

In the ***app.component.ts*** file, after the last modification we add:

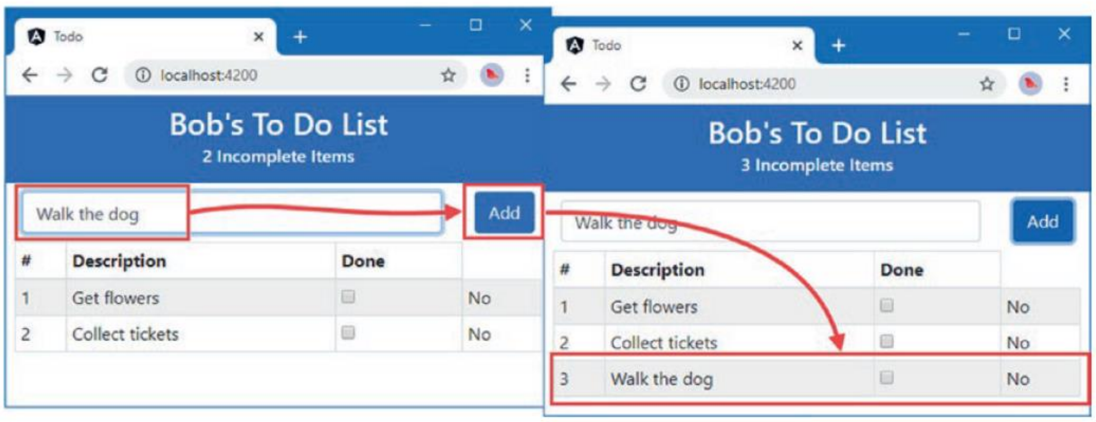
```
  get items(): readonly TodoItem[] {
    return this.list.items.filter(item => !item.complete);
  }

  addItem(newItem) {
    if (newItem != "") {
          this.list.addItem(newItem);
      }
    }
 }
```

Event Binding

```
...
<input class="form-control" placeholder="Enter task here" #todoText />
...
```

Variable

```
        ...
        <button class="btn btn-primary mt-1" (click)="addItem(todoText.value)">
        ...
```



Bob's To Do List
2 Incomplete Items

Walk the dog — Add

#	Description	Done	
1	Get flowers	☐	No
2	Collect tickets	☐	No

Bob's To Do List
3 Incomplete Items

Walk the dog — Add

#	Description	Done	
1	Get flowers	☐	No
2	Collect tickets	☐	No
3	Walk the dog	☐	No

2.9) Mostramos ítems completados: This is done by removing the Yes/No column in the table from the template and adding the option to show completed tasks:

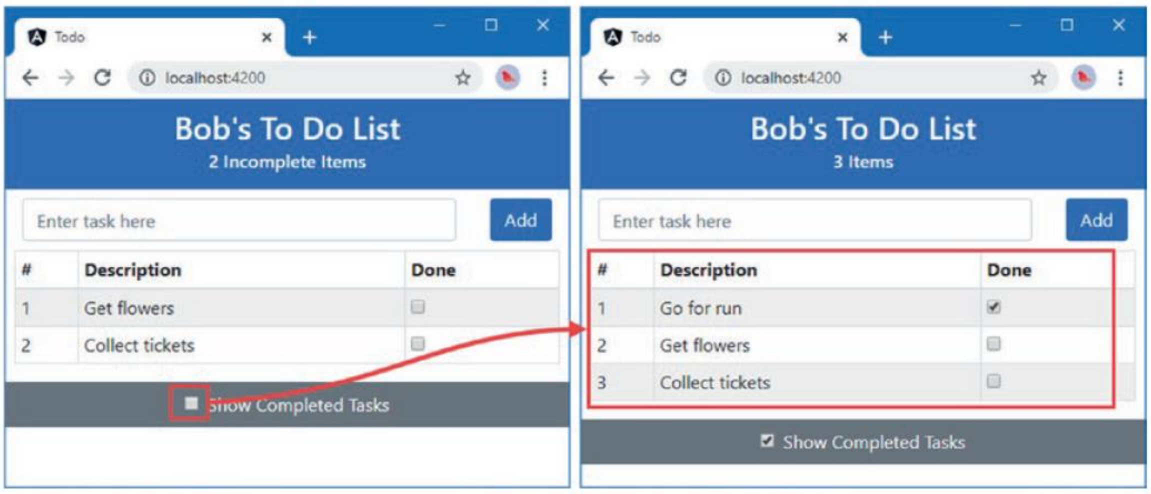
```
<h3 class="bg-primary text-center text-white p-2">
  {{ username }}'s To Do List
  <h6 class="mt-1">{{ itemCount }} {{ showComplete ? "" : "Incomplete" }} Items</h6>
</h3>

<div class="container-fluid">
  <div class="row">
    <div class="col">
      <input class="form-control" placeholder="Enter task here" #todoText />
    </div>
    <div class="col-auto">
      <button class="btn btn-primary" (click)="addItem(todoText.value)">
        Add
      </button>
    </div>
  </div>
</div>
```

```
<div class="m-2">
  <table class="table table-striped table-bordered table-sm">
    <thead>
        <tr><th>#</th><th>Description</th><th>Done</th></tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of items; let i = index">
            <td>{{ i + 1 }}</td>
            <td>{{ item.task }}</td>
            <td><input type="checkbox" [(ngModel)]="item.complete" /></td>
            <!-- <td [ngSwitch]="item.complete">
                <span *ngSwitchCase="true">Yes</span>
                <span *ngSwitchDefault>No</span>
            </td> -->
        </tr>
    </tbody>
  </table>
</div>

<div class="bg-secondary text-white text-center p-2">
  <div class="form-check">
    <input class="form-check-input" type="checkbox" [(ngModel)]="showComplete" />
    <label class="form-check-label" for="defaultCheck1">
      Show Completed Tasks
    </label>
  </div>
</div>
```

In the ***app.component.ts*** file in the last part we add:

```
  get itemCount(): number {
    return this.items.length;
  }

  get items(): readonly TodoItem[] {
    return this.list.items.filter(item => this.showComplete || !item.complete);
  }

  addItem(newItem) {
    if (newItem != "") {
        this.list.addItem(newItem);
    }
  }

  showComplete: boolean = false;
}
```

Se we can decide if we want to see completed tasks or not:



#	Description	Done
1	Get flowers	☐
2	Collect tickets	☐

#	Description	Done
1	Go for run	✓
2	Get flowers	☐
3	Collect tickets	☐

19.7.20 – Probamos en Probook / UB18.04: hacer download del repo en Miot2020/Materias/DAM/libro (aprox), luego abrir desde VSC el directorio. En el repo no están todos los /node_modules (aprox 130MB por cada ejercicio), así que hay que descargarlos, en Ej2 por ejemplo se entra a `cd todo` , y luego `npm install` (tarda un buen rato, en

descargar todos los modulos de node). Finalmente con `ng serve` se corre el compilador, y queda listo para correr en el Chrome haciendo http://localhost:4200.
Finalmente, ensayamos copiando el Ejercicio 2 completo a la carpeta /Practica/delLibro/Ej2_modificado. Tocamos las partes específicas del app.component.html (ej.: Rulo's Lista de Tareas")- dentro de `cd todo`. Finalmente hay que volver a correr `npm install` (a pesar de que habíamos copiado todos los modulos de node). Finalmente con `ng serve` se corre el compilador, y corre en el Chrome haciendo http://localhost:4200.

3) Páginas Web Clásicas (RoundTrip), SPAs (Single Page Apps) y Modelo MVC:

La página web tradicional y muchas aplicaciones GUI desde las de Xerox PARC en adelante se construyeron sobre el concepto de Model-View-Controller que realiza una separación de roles entre los datos y la lógica. En las páginas web Roundtrip el browser sólo envía consultas HTTP y espera contenido, toda la lógica está del lado del servidor como se ve en Figure 3.1

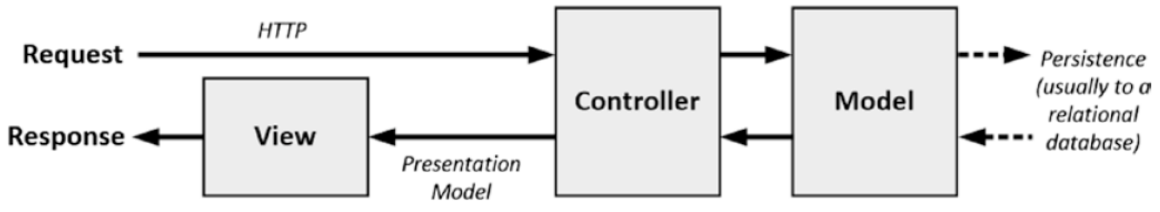


Figure 3-1. The server-side implementation of the MVC pattern

Dice Freeman:
*The key to applying the MVC pattern is to implement the key premise of a separation of concerns, in which the data model in the application is decoupled from the business and presentation logic. In client-side web development, this means separating the **data**, the **logic** that operates on that data, and the **HTML elements used to display the data**. The result is a client-side application that is easier to develop, maintain, and test.*
…The client-side implementation of the MVC pattern gets its data from server-side components, usually via a RESTful web service. The goal of the controller and the view is to operate on the data in the model to perform DOM manipulation so as to create and manage HTML elements that the user can interact with. Those interactions are fed back to the controller, closing the loop to form an interactive application (Figure 3.2)
Angular utiliza los terminos

1) Template para el bloque View → ***app.component.html***
2) Component para Controller → ***app.component.ts***

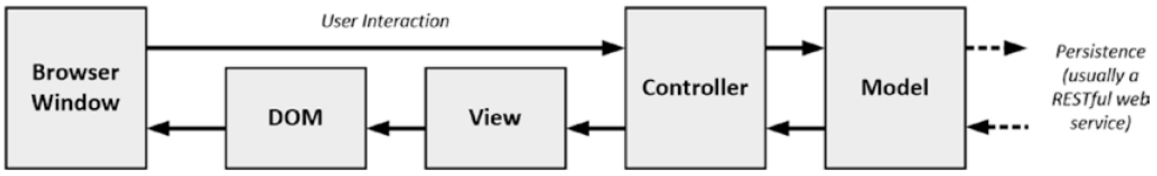


Figure 3-2. A client-side implementation of the MVC pattern

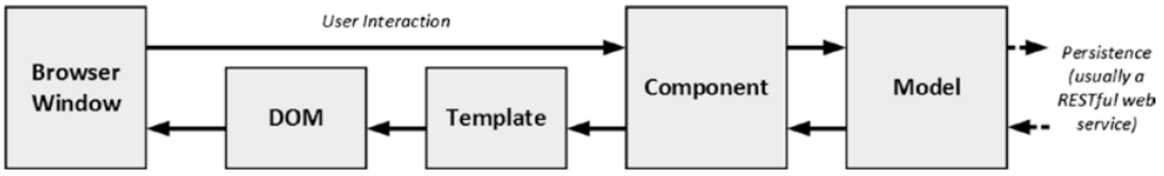


Figure 3-3. The Angular implementation of the MVC pattern

Dice Freeman:
***Models**—the M in MVC—contain the data that users work with. There are two broad types of model: **view models**, which represent just data passed from the component to the template, and **domain models**, which contain the data in a business domain, along with the operations, transformations, and rules for creating, storing, and manipulating that data, collectively referred to as the model logic.*

.. The benefits of ensuring that the model is isolated from the controller and views are that you can test your logic more easily (→Angular unit testing: Chapter 29) and that enhancing/maintaining the overall application is simpler and easier.
.. The best domain models contain the logic for getting and storing data persistently and contain the logic for create, read, update, and delete operations (CRUD)→ access to databases.

***..Controllers**, which are known as components in Angular, are the connective tissue in an Angular web app; they act as conduits between the data model and views.*
A component (*app.component.ts*) that follows the MVC pattern should*:*

- ***Contain the logic required to set up the initial state of the template***
- ***Contain the logic/behaviors required by the template to present data from the model***
- ***Contain the logic/behaviors required to update the model based on user interaction***

A component should not:

- ***Contain logic that manipulates the DOM (that is the job of the template→ app.component.html)***
- ***Contain logic that manages the persistence of data (that is the job of the model→DB access)***

***..Views**, which are known as templates in Angular, are defined using HTML elements (**app.component.html)** that are enhanced by **data bindings**. It is the data bindings that make Angular so flexible, and they transform HTML elements into the foundation for dynamic web applications.*
Templates should:

- ***Contain the logic and markup required to present data to the user***

Templates should not:

- ***Contain complex logic (this is better placed in a component or one of the other Angular building blocks, such as directives, services, or pipes)***
- ***Contain logic that creates, stores, or manipulates the domain model***

Templates can contain logic, but it should be simple and used sparingly. Putting anything but the simplest method calls or expressions in a template makes the overall application harder to test and maintain.

3.1) RESTFull services - Páginas

*The logic for domain models in Angular apps is often split between the client and the server. The **server contains the persistent store**, typically a database, and contains the logic for managing it. You don't want the client-side code accessing the data store directly—doing so would create a tight coupling between the client and the data store that would complicate unit testing and make it difficult to change the data store without also making changes to the client code.*
In a RESTful web service, the operation that is being requested is expressed through a combination of the HTTP method and the URL. There is no standard URL specification for a RESTful web service, but the idea is to make the URL self-explanatory, such that it is obvious what the URL refers to. → usa el ejemplo:

```
http://myserver.mydomain.com/people/bob
```

The URL identifies the data object that I want to operate on, and the HTTP method specifies what operation I want to be performed:

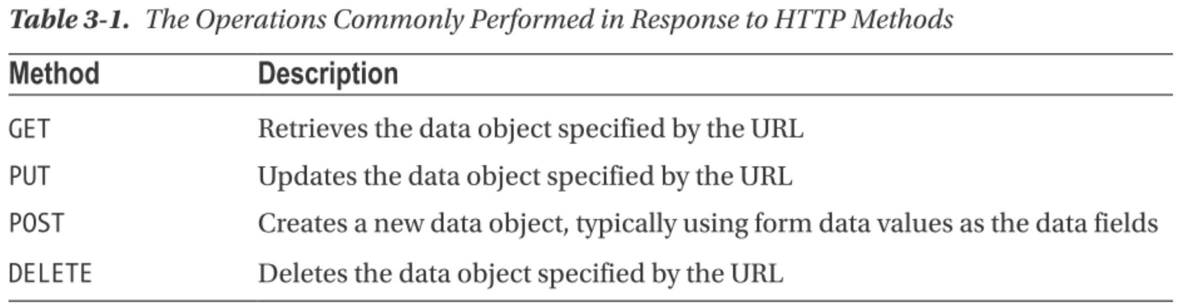


***Table 3-1.** The Operations Commonly Performed in Response to HTTP Methods*

Method	Description
GET	Retrieves the data object specified by the URL
PUT	Updates the data object specified by the URL
POST	Creates a new data object, typically using form data values as the data fields
DELETE	Deletes the data object specified by the URL

You don't have to use the HTTP methods to perform the operations I describe in the table. A common variation is that the POST method is often used to serve double duty and will update an object if one exists and create one if not, meaning that the PUT method isn't used.

3.2) Logic Placement: *Knowing where to put logic becomes second nature as you get more experience in Angular development, but here are the three rules:*

- ***Template logic should prepare data only for display and never modify the model.***
- ***Component logic should never directly create, update, or delete data from the model.***
- ***The templates and components should never directly access the data store.***

20.7.2020

4) HTML /CSS Primer:

4.1) Página muy básica de repaso, crea un archivo package.json para definir las dependencias:

```
{
  "dependencies": {
    "bootstrap": "4.4.1"
  }
}
```

para que incluya lo requerido de CSS / bootstrap, hay que correr

```
 npm install
```

4.2) Página de armado de la Lista de Tareas

```
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Tareas</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />
</head>
<body class="m-1">
    <h3 class="bg-primary text-white p-3">Rulo Lista de Tareas</h3>
    <div class="my-1">
        <input class="form-control" />
        <button class="btn btn-primary mt-1">Agregar</button>
    </div>
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Descripcion</th>
                <th>Terminado</th>
            </tr>
        </thead>
        <tbody>
            <tr><td>Comprar comida</td><td>No</td></tr>
            <tr><td>Pasear a Lana</td><td>No</td></tr>
            <tr><td>Sacar la basura</td><td>Yes</td></tr>
            <tr><td>Llamar al gasista</td><td>No</td></tr>
        </tbody>
    </table>
</body>
</html>
```

4.3) Para correrla, y que cada vez que se cambie se actualice, se puede usar:

```
npx lite-server@2.5.4
```

y si no abre el browser por defecto se puede ver en http://localhost:3000

4.4) Repaso → 4.4.1) elemento de celda:

Start Tag	***Contenido***	***End Tag***
`<td>`	`Comprar comida`	`</td>`

4.4.2) Los llamados **`void or self-closing elements`**,

```
and they are written without a separate end tag, like this:
...
<input />
```

The input element is the most used void element, and its purpose is to allow the user to provide input, through a text field, radio button, or checkbox.

4.4.3) Atributos*(i)*

...

```
<link href="node_modules/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
```

...

This is a **link** *element, and it imports content into the document. There are two attributes, href and rel*
. Attributes are always defined as part of the start tag, and these
attributes have a name and a value.

1. ***href*** *attribute specifies the content to import,*
2. ***rel*** *attribute tells the browser what kind of content it is.*

The attributes on this link element tell the browser to ***import the bootstrap.min.css*** *file and to* ***treat it as a style sheet****, which is a file that contains CSS styles.*

4.4.4) Atributos*(ii) sin valores – le dice al browser que espera cierto comportamiento*

...

```
<input class="form-control" required />
```

...

4.4.5) Atributos*(iii) con i) variables → usado en Angular*

Angular relies on HTML element attributes to apply a lot of its functionality. Most of the time, the values of attributes are evaluated as JavaScript expressions, such as with this element:

...

```
<td [ngSwitch]="item.complete">...
```

The attribute applied to the ***td*** *element tells Angular to read the value of a property called* **complete** *on an object that has been assigned to a variable called* **item***.*

ii) "literal values" *to tell Angular that it is dealing with a literal value, like this:*

...

```
<td [ngSwitch]="'Apples'">
```

...

The attribute value contains the string Apples, which is quoted in both single and double quotes.

21.07.20

4.4.6) Estructura jerarquica

```
<thead>
    <tr>
        <th>Description</th>
        <th>Done</th>
    </tr>
</thead>
...
```

In the listing, the `thead` *element contains* `tr` *elements that, in turn, contain* `th` *elements. Arranging elements is a key concept in HTML because it imparts the significance of the outer element to those contained within.*

4.4.7) Elementos HTML usados en Ej2.

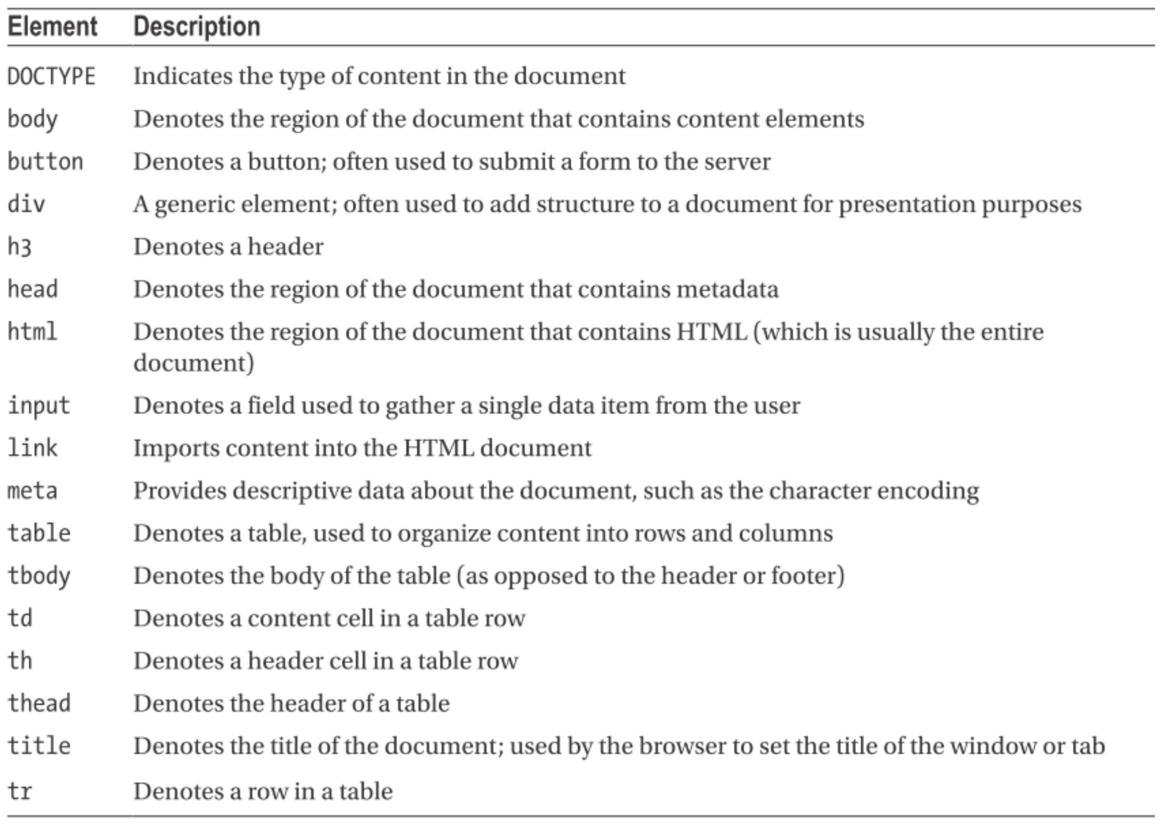


Table 4-1. *HTML Elements Used in the Example Document*

Element	Description
`DOCTYPE`	Indicates the type of content in the document
`body`	Denotes the region of the document that contains content elements
`button`	Denotes a button; often used to submit a form to the server
`div`	A generic element; often used to add structure to a document for presentation purposes
`h3`	Denotes a header
`head`	Denotes the region of the document that contains metadata
`html`	Denotes the region of the document that contains HTML (which is usually the entire document)
`input`	Denotes a field used to gather a single data item from the user
`link`	Imports content into the HTML document
`meta`	Provides descriptive data about the document, such as the character encoding
`table`	Denotes a table, used to organize content into rows and columns
`tbody`	Denotes the body of the table (as opposed to the header or footer)
`td`	Denotes a content cell in a table row
`th`	Denotes a header cell in a table row
`thead`	Denotes the header of a table
`title`	Denotes the title of the document; used by the browser to set the title of the window or tab
`tr`	Denotes a row in a table

4.4.8) DOM or Document Object Model
When the browser loads and processes an HTML document, it creates the Document Object Model (DOM). the DOM is a model in which JavaScript objects are used to represent each element in the document, and the DOM is the mechanism by which you can programmatically engage with the content of an HTML document. You rarely work directly with the DOM in Angular, but it is important to understand that the browser maintains a live model of the HTML document represented by JavaScript objects. When Angular modifies these objects, the browser updates the content it displays to reflect the modifications. This is one of the key foundations of web applications. If we were not able to modify the DOM, we would not be able to create client-side web apps.

4.4.9) BOOTSTRAP y CSS
HTML elements tell the browser what kind of content they represent, but they don't provide any information about how that content should be displayed. The information about how to display elements is provided using Cascading Style Sheets (CSS). CSS consists of properties that can be used to configure every aspect of an element's appearance and selectors that allow those properties to be applied. CSS is flexible and powerful, but it requires time and close attention to detail to get good, consistent results, especially as some legacy browsers implement features inconsistently. CSS frameworks provide a set of styles that can be easily applied to produce consistent effects throughout a project.

The most widely used framework is Bootstrap **(parecido al Materialize que usamos en DAW)***, which consists of CSS classes that can be applied to elements to style them consistently and JavaScript code that performs additional enhancements. I use the Bootstrap CSS styles in this book because they let me style my examples without having to define custom styles in each chapter. I don't use the Bootstrap JavaScript features at all in this book since the interactive parts of the examples are provided using Angular.*

See http://getbootstrap.com for full details of the features.

4.4.10) Aplicación de BOOTSTRAP en Ej.2: *Bootstrap styles are applied via the class attribute, which is used to group related elements. The class attribute isn't just used to apply CSS styles, but it is the most common use, and it underpins the way that Bootstrap and similar frameworks operate. Here is an HTML element with a class attribute, taken from the index.html file:*

```
...
<button class="btn btn-primary mt-1">Add</button>
...
```

The class attribute assigns the button element to three classes, whose names are separated by spaces:
btn, btn-primary, *and* mt-1*. These classes correspond to styles defined by Bootstrap:*

Table 4-2. *The Three Button Element Classes*

Name	Description
btn	This class applies the basic styling for a button. It can be applied to button or a elements to provide a consistent appearance.
btn-primary	This class applies a style context to provide a visual cue about the purpose of the button. See the "Using Contextual Classes" section.
mt-1	This class adds a gap between the top of the element and the content that surrounds it. See the "Using Margin and Padding" section.

4.4.10a) Bootstrap y las clases contextuales: Bootstrap defines a set of style contexts that are used to style related elements consistently. These contexts, which are described in Table 4-3, are used in the names of the classes that apply Bootstrap styles to elements.

Table 4-3. *The Bootstrap Style Contexts*

Name	Description
primary	This context is used to indicate the main action or area of content.
secondary	This context is used to indicate the supporting areas of content.
success	This context is used to indicate a successful outcome.
info	This context is used to present additional information.
warning	This context is used to present warnings.
danger	This context is used to present serious warnings.
muted	This context is used to de-emphasize content.
dark	This context is used to increase contrast by using a dark color.
white	This context is used to increase contrast by using white.

Ejemplo de primary context en Ej2: bg indica background.

```
...
<h3 class="bg-primary text-white p-3">Adam's To Do List</h3>
...
<button class="btn btn-primary mt-1">Add</button>
...
```

4.4.10b) Padding / margin

Bootstrap includes some utility classes that are used to add padding (space between an element's inner edge and its content) and margin (space between an element's edge and the surrounding elements). The benefit of using these classes is that they apply a consistent amount of spacing throughout the application. The names of these classes follow a well-defined pattern. Here is the body element from the index.html file created at the start of the chapter, to which margin has been applied:

```
...
<body class="m-1">
...
```

The classes that apply margin and padding to elements follow a well-defined naming schema: first, the letter m (for margin) or p (for padding), then a hyphen, and then a number indicating how much space should be applied (0 for no spacing, or 1, 2, or 3 for increasing amounts). You can also add a letter to apply spacing only to specific sides, so t for top, b for bottom, l for left, r for right, x for left and right, and y for top and bottom.

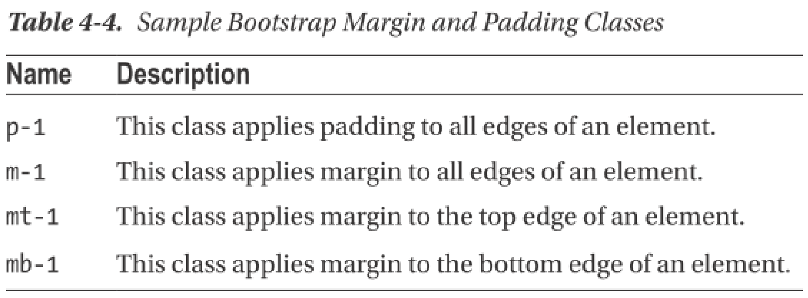


Table 4-4. *Sample Bootstrap Margin and Padding Classes*

Name	Description
`p-1`	This class applies padding to all edges of an element.
`m-1`	This class applies margin to all edges of an element.
`mt-1`	This class applies margin to the top edge of an element.
`mb-1`	This class applies margin to the bottom edge of an element.

Resto de CAP 4. → muy dedicado a similaridades con Materialize...

21.7.2020 (cont)

5) Javascript Primer: Chapter 5 uses Angular to show some Javascript basics, by including them in an empty typescript file (main.ts). Much of it is very basic, but some issues about functions are useful.

5.1) Javascript functions: *Using Default and Rest Parameters*

The number of arguments you provide when you invoke a function doesn't need to match the number of parameters in the function. If you call the function with fewer arguments than it has parameters, then the value of any parameters you have not supplied values for is undefined, which is a special JavaScript value.

If you call the function with more arguments than there are parameters, then the additional arguments are ignored. The consequence of this is that you can't create two functions with the same name and different parameters and expect JavaScript to differentiate between them based on the arguments you provide when invoking the function.

This is called polymorphism, and although it is supported in languages such as Java and C#, it isn't available in JavaScript. Instead, if you define two functions with the same name, then the second definition replaces the first.

There are two ways that you can modify a function to respond to a mismatch between the number of parameters it defines and the number of arguments used to invoke it:
*5.1.1) **Default parameters** deal with the situation where there are fewer arguments than parameters, and they allow you to provide a default value for the parameters for which there are no arguments, as shown in: (Listing 5-8.)*

```
let myFunc = function (name, weather = "raining") {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today");
};
myFunc("Adam");
```

The weather parameter in the function has been assigned a default value of raining, which will be used if the function is invoked with only one argument, producing the following results:

```
Hello Adam.
It is raining today
```

*5.1.2) **Rest parameters** are used to capture any additional arguments when a function is invoked with additional arguments, as shown in: (Listing 5-9)*

```
let myFunc = function (name, weather, ...extraArgs) {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today");
    for (let i = 0; i < extraArgs.length; i++) {
        console.log("Extra Arg: " + extraArgs[i]);
    }
};
myFunc("Adam", "sunny", "one", "two", "three");
```

The rest parameter must be the last parameter defined by the function, and its name is prefixed with an ellipsis (three periods, ...). The rest parameter is an array to which any extra arguments will be assigned. In the listing, the function prints out each extra argument to the console, producing the following results:

```
Hello Adam.
It is sunny today
Extra Arg: one
Extra Arg: two
Extra Arg: three
```

5.2) Javascript functions: Defining Functions That Return Results
You can return results from functions using the return keyword. Listing 5-10 shows a function that returns a Result, in main.ts

```
let myFunc = function(name) {
    return ("Hello " + name + ".");
};
console.log(myFunc("Adam"));
```

Output:
```
Hello Adam.
```

5.3) Javascript functions: Using Functions as arguments to other functions.
JavaScript functions can be passed around as objects, which means you can use one function as the argument to another, as demonstrated in Listing 5-11.

```
let myFunc = function (nameFunction) {
    return ("Hello " + nameFunction() + ".");
};
console.log(myFunc(function () {
    return "Adam";
}));
```

The `myFunc` *function defines a parameter called* `nameFunction` *that it invokes to get the value to insert into the string that it returns. I pass a function that returns Adam as the argument to* `myFunc`*, which produces the following output:*

```
Hello Adam.
```

5.4) Chaining Javascript functions:
(Listing 5.12)
```
let myFunc = function (nameFunction) {
    return ("Hello " + nameFunction() + ".");
};
let printName = function (nameFunction, printFunction) {
    printFunction(myFunc(nameFunction));
}
printName(function () { return "Adam" }, console.log);
```

Also produces:

```
Hello Adam
```

5.5) Fat Arrow Functions:

*Arrow functions—also known as **fat arrow functions or lambda expressions**—are an alternative way of defining functions and are often used to **define functions that are used only as arguments** to other functions.*
(Listing 5-13 replaces the functions from the previous example 5.12 with arrow functions)

```
let myFunc = function (nameFunction) {
    return  ("Hello  "  +  nameFunction()  +
".");
};
```

```
let printName = function (nameFunction, printFunction)
{
    printFunction(myFunc(nameFunction));
}
```

```
let myFunc = (nameFunction) => ("Hello " + nameFunction() + ".");

let printName = (nameFunction, printFunction) => printFunction(myFunc(nameFunction));

printName(function () { return "Adam" }, console.log);
```

These functions perform the same work as the ones in Listing 5-12. There are three parts to an arrow function: the input parameters, then an equal sign and a greater-than sign (the "arrow"), and finally the function result. The return keyword and curly braces are required only if the arrow function needs to execute more than one statement. There are more examples of arrow functions later in this chapter.

5.6) Variable scope and type

*The **let** keyword is used to declare variables and, optionally, assign a value to the variable in a single statement. Variables declared with let are scoped to the region of code in which they are defined (Listing 5-14).*

```
let messageFunction = function (name, weather) {
    let message = "Hello, Adam";
    if (weather == "sunny") {
        let message = "It is a nice day";
        console.log(message);
    } else {
        let message = "It is " + weather + " today";
        console.log(message);
    }
    console.log(message);
}
messageFunction("Adam", "raining");
```

In this example, there are three statements that use the let keyword to define a variable called message. The scope of each variable is limited to the region of code that it is defined in, producing the following results:

```
It is raining today
Hello, Adam
```

This may seem like an odd example, but there is another keyword that can be used to declare variables:
***var.** The **let** keyword is a relatively new addition to the JavaScript specification that is intended to address some oddities in the way **var** behaves. Listing 5-15 takes the example from Listing 5-14 and replaces **let** with **var**.*

```
let messageFunction = function (name, weather) {
    var message = "Hello, Adam";
    if (weather == "sunny") {
        var message = "It is a nice day";
        console.log(message);
    } else {
        var message = "It is " + weather + " today";
        console.log(message);
    }
    console.log(message);
}
messageFunction("Adam", "raining");
```
When you save the changes in the listing, you will see the following results:

```
It is raining today
It is raining today
```

*The problem is that the **var** keyword creates variables whose scope is the containing function, which means that all the references to message are referring to the same variable. This can cause unexpected results for even experienced JavaScript developers and is the reason that the more conventional **let** keyword was introduced.*

5.7) Using Variable Closure

*If you define a function inside another function—creating inner and outer functions—then the inner function is able to access the variables of the outer function, using a feature called **closure**, as demonstrated in Listing 5-16.*

```
let myFunc = function(name) {
    let myLocalVar = "sunny";
    let innerFunction = function () {
        return ("Hello " + name + ". Today is " + myLocalVar + ".");
    }
    return innerFunction();
};
console.log(myFunc("Adam"));
```

The inner function in this example is able to access the local variables of the outer function, including its name parameter. This is a powerful feature that means you don't have to define parameters on inner functions to pass around data values, but caution is required because it is easy to get unexpected results when using common variable names like counter or index, where you may not realize that you are reusing a variable name from the outer function. This example produces the following results:

```
Hello Adam. Today is sunny.
```

5.8) Working with strings

***5.8.1)** You define string values using either the double-quote or single quote characters, as shown in Listing 5-18.*

```
let firstString = "This is a string";
let secondString = 'And so is this';
```

The quote characters you use must match. You can't start a string with a single quote and finish with a double quote.

***5.8.2)** Typical use como en el ejemplo anterior 5.16 ...*`let message = "It is " + weather + " today";`

***5.8.3)** Template Strings: Lo anterior se puede reemplazar por:*

```
let messageFunction = function (weather) {
    let message = `It is ${weather} today`;
    console.log(message);
}
messageFunction("raining");
```

Template strings begin and end with backticks (the ` character), and data values are denoted by curly braces preceded by a dollar sign. This string, for example, incorporates the value of the weather variable into
the template string,and produces the following output:

```
It is raining today
```

5.8.4) *Table of string properties and methods*

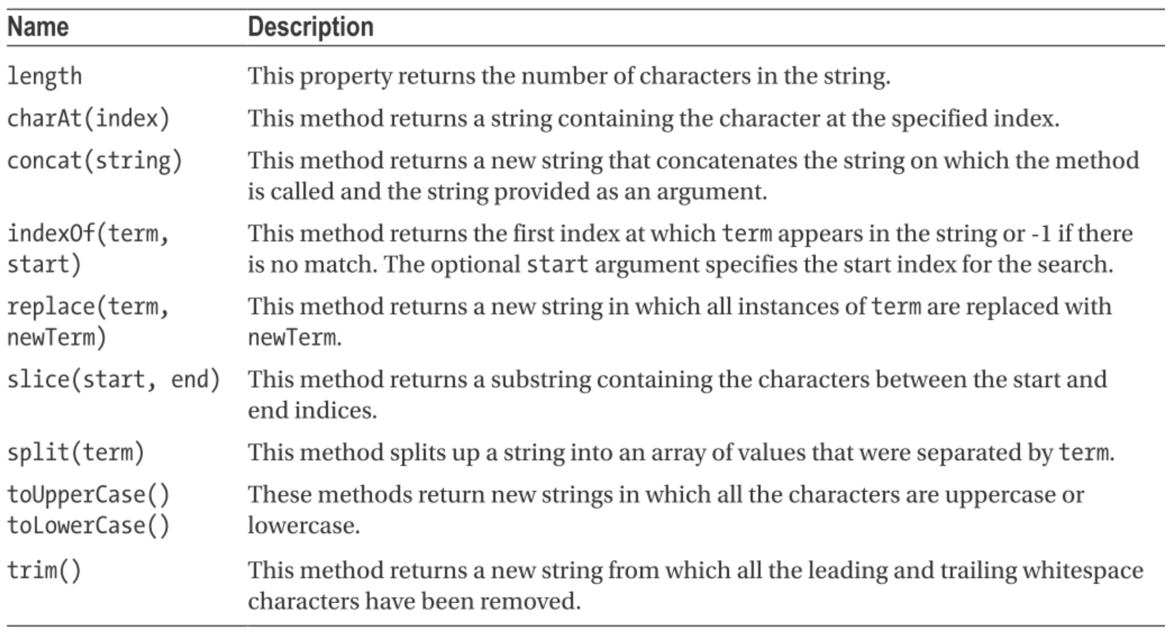


Table 5-2. Useful string Properties and Methods

Name	Description
`length`	This property returns the number of characters in the string.
`charAt(index)`	This method returns a string containing the character at the specified index.
`concat(string)`	This method returns a new string that concatenates the string on which the method is called and the string provided as an argument.
`indexOf(term, start)`	This method returns the first index at which `term` appears in the string or -1 if there is no match. The optional `start` argument specifies the start index for the search.
`replace(term, newTerm)`	This method returns a new string in which all instances of `term` are replaced with `newTerm`.
`slice(start, end)`	This method returns a substring containing the characters between the start and end indices.
`split(term)`	This method splits up a string into an array of values that were separated by `term`.
`toUpperCase()` `toLowerCase()`	These methods return new strings in which all the characters are uppercase or lowercase.
`trim()`	This method returns a new string from which all the leading and trailing whitespace characters have been removed.

5.9) **Identity operator**

If you want to test to ensure that the values and the types are the same, then you need to use the identity operator (===, three equal signs, rather than the two of the equality operator), as shown in Listing 5-23.

```
let firstVal = 5;
let secondVal = "5";
if (firstVal === secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

In this example, the identity operator will consider the two variables to be different. This operator doesn't coerce types. The result from this script is as follows:

```
They are NOT the same
```

***5.10) Add* Operator Precedence: Ojo!** *The string concatenation operator (+) has higher precedence than the addition operator (also +), which means JavaScript will concatenate variables in preference to adding.*

```
let myData1 = 5 + 5;
let myData2 = 5 + "5";
console.log("Result 1: " + myData1);
console.log("Result 2: " + myData2);
```

The result from this script is as follows:

```
Result 1: 10
Result 2: 55
```

It also means that the listing:

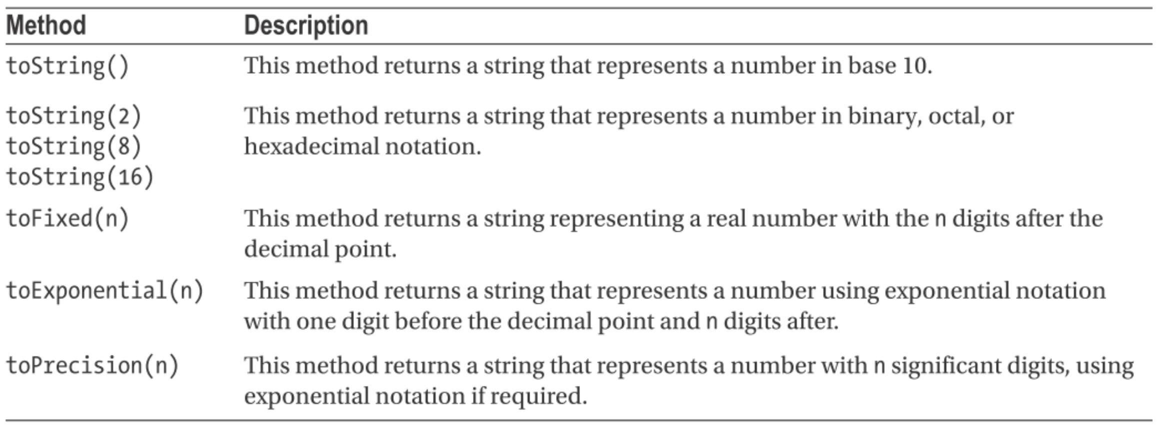
```
let myData1 = (5).toString() + String(5);
console.log("Result: " + myData1);
```

Produces:

```
Result: 55
```

5.11) Useful number to string methods table

Table 5-4. Useful Number-to-String Methods



Method	Description
`toString()`	This method returns a string that represents a number in base 10.

`toString(2)`
`toString(8)`
`toString(16)`
	This method returns a string that represents a number in binary, octal, or hexadecimal notation.

`toFixed(n)`	This method returns a string representing a real number with the n digits after the decimal point.
`toExponential(n)`	This method returns a string that represents a number using exponential notation with one digit before the decimal point and n digits after.
`toPrecision(n)`	This method returns a string that represents a number with n significant digits, using exponential notation if required.

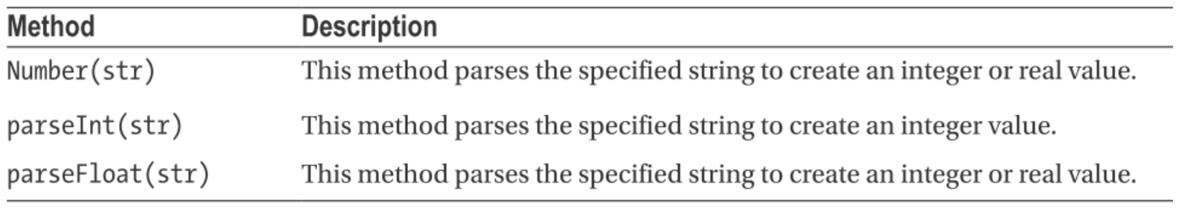
5.12) String to number conversion

```
let firstVal = "5";
let secondVal = "5";
let result = Number(firstVal) + Number(secondVal);
console.log("Result: " + result);
```

5.12a) Useful string to number methods table (5.5)

The Number function is strict in the way that it parses string values, but there are two other functions you can use that are more flexible and will ignore trailing non-number characters. These functions are parseInt and parseFloat.

Table 5-5. Useful String to Number Methods



Method	Description
`Number(str)`	This method parses the specified string to create an integer or real value.
`parseInt(str)`	This method parses the specified string to create an integer value.
`parseFloat(str)`	This method parses the specified string to create an integer or real value.

5.13) Array definition and enumeration.

Arrays can be of mixed types, and are initiliazed with new as in

```
let myArray = new Array();
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

Or more conveniently with []. Listing is as in conventional languages.

```
let myArray = [100, "Adam", true];
for (let i = 0; i < myArray.length; i++) {
    console.log("Index " + i + ": " + myArray[i]);
}
console.log("---");
// Equivalent to..
myArray.forEach((value, index) => console.log("Index " + index + ": " + value));
```

The JavaScript for loop works just the same way as loops in many other languages. You determine how many elements there are in the array by using the length property.
The function passed to the forEach method is given two arguments: the **value** *of the current item to be processed and the position of that item in the array. In this listing, I have used an arrow function as the argument to the forEach method, which is the kind of use for which they excel (and you will see used throughout this book). The output from the listing is as follows:*

```
Index 0: 100
Index 1: Adam
Index 2: true
---
Index 0: 100
Index 1: Adam
Index 2: true
```

5.14) Spread (… ellipsis) Array operator

The JavaScript … spread or ellipsis operator "unpacks" the first (myArray) into the second (otherArray)

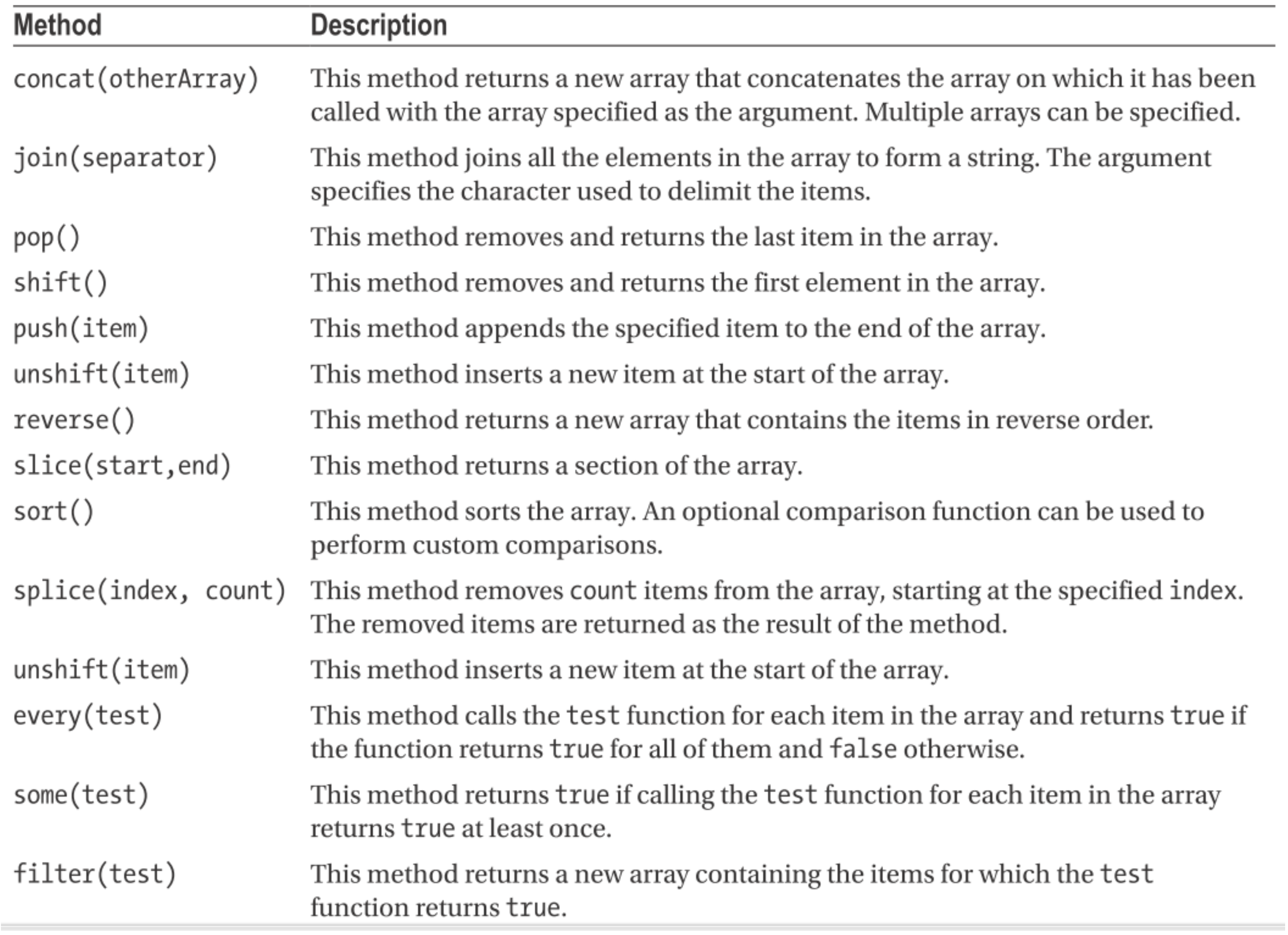
```
let myArray = [100, "Adam", true];
let otherArray = [...myArray, 200, "Bob", false];
for (let i = 0; i < otherArray.length; i++) {
  console.log(`Array item ${i}: ${otherArray[i]}`);
}
```

The Output is:

```
Array item 0: 100
Array item 1: Adam
Array item 2: true
Array item 3: 200
Array item 4: Bob
Array item 5: false
```

5.15) Table of array Methods

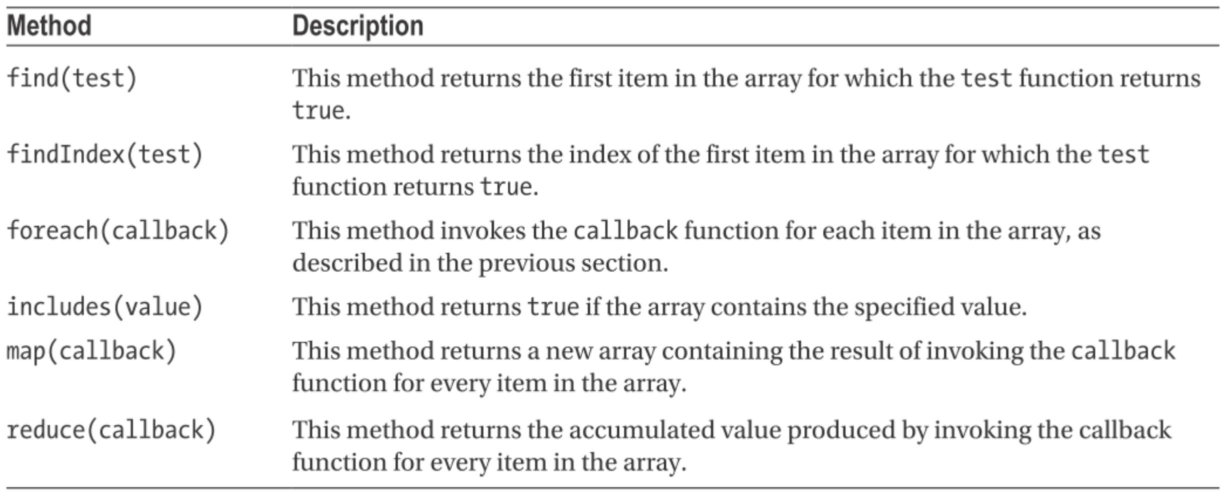
***Table 5-6.** Useful Array Methods*



Method	Description
`concat(otherArray)`	This method returns a new array that concatenates the array on which it has been called with the array specified as the argument. Multiple arrays can be specified.
`join(separator)`	This method joins all the elements in the array to form a string. The argument specifies the character used to delimit the items.
`pop()`	This method removes and returns the last item in the array.
`shift()`	This method removes and returns the first element in the array.
`push(item)`	This method appends the specified item to the end of the array.
`unshift(item)`	This method inserts a new item at the start of the array.
`reverse()`	This method returns a new array that contains the items in reverse order.
`slice(start,end)`	This method returns a section of the array.
`sort()`	This method sorts the array. An optional comparison function can be used to perform custom comparisons.
`splice(index, count)`	This method removes `count` items from the array, starting at the specified `index`. The removed items are returned as the result of the method.
`unshift(item)`	This method inserts a new item at the start of the array.
`every(test)`	This method calls the `test` function for each item in the array and returns `true` if the function returns `true` for all of them and `false` otherwise.
`some(test)`	This method returns `true` if calling the `test` function for each item in the array returns `true` at least once.
`filter(test)`	This method returns a new array containing the items for which the `test` function returns `true`.

(continued next page)

Table 5-6. (*continued*)



Method	Description
find(test)	This method returns the first item in the array for which the test function returns true.
findIndex(test)	This method returns the index of the first item in the array for which the test function returns true.
foreach(callback)	This method invokes the callback function for each item in the array, as described in the previous section.
includes(value)	This method returns true if the array contains the specified value.
map(callback)	This method returns a new array containing the result of invoking the callback function for every item in the array.
reduce(callback)	This method returns the accumulated value produced by invoking the callback function for every item in the array.

Since many of the methods in Table 5-6 return a new array, these methods can be chained together to process a filtered data array, as shown:.
Listing 5-33. Processing a Data Array

```
let products = [
    { name: "Hat", price: 24.5, stock: 10 },
    { name: "Kayak", price: 289.99, stock: 1 },
    { name: "Soccer Ball", price: 10, stock: 0 },
    { name: "Running Shoes", price: 116.50, stock: 20 }
];

let totalValue = products
    .filter(item => item.stock > 0)
    .reduce((prev, item) => prev + (item.price * item.stock), 0);
console.log("Total value: $" + totalValue.toFixed(2));
```

I use the filter method to select the items in the array whose stock value is greater than zero and use the reduce method to determine the total value of those items, producing the following output:

```
Total value: $2864.99
```

6) Javascript & Typescript Additionals:

6.1) Objects in JS/TS:
6.1.1) Creating an Object in the main.ts File in the src Folder

```
let myData = new Object();
myData.name = "Adam";
myData.weather = "sunny";
console.log("Hello " + myData.name + ".");
console.log("Today is " + myData.weather + ".");
```

I create an object by calling new Object(), and I assign the result (the newly created object) to a variable called myData. Once the object is created, I can define properties on the object just by assigning values, as in: `myData.name = "Adam";` *Prior to this statement, my object doesn't have a property called name. When the statement has executed, the property does exist, and it has been assigned the value Adam.*
Output is:

```
Hello Adam.
Today is sunny.
```

6.1.2) Creating an Object Using the Object Literal Format

```
let myData = {
    name: "Adam",
    weather: "sunny"
};
console.log("Hello " + myData.name + ". ");
console.log("Today is " + myData.weather + ".");
```

Each property that you want to define is separated from its value using a colon (:), and properties are separated using a comma (,). The effect is the same as in the previous example, and the result from the listing is as follows:

```
Hello Adam.
Today is sunny.
```

6.2) Using Functions as Methods:

*One of the best features like most about JavaScript is the way you can add functions to objects. A function defined on an object is called a **method**. Listing 6-3 shows how you can add methods in this manner.*

```
let myData = {
    name: "Adam",
    weather: "sunny",
    printMessages: function () {
        console.log("Hello " + this.name + ". ");
        console.log("Today is " + this.weather + ".");
    }
};
myData.printMessages();
```

*In this example, I have used a function to create a method called **printMessages**. Notice that to refer to the properties defined by the object, I have to use the **this** keyword. When a function is used as a method, the function is implicitly passed the object on which the method has been called as an argument through the special variable **this**. The output from the listing is as follows:*

```
Hello Adam.
Today is sunny.
```

6.3) Defining Classes

*Classes are templates that are used to create **objects that have identical functionality**. Support for classes is a recent addition to the JavaScript specification and is intended to make working with JavaScript more consistent with other mainstream programming languages, and classes are used throughout Angular development. Listing 6-4 shows how the functionality defined by the object in the previous section can be expressed using a class.*

```
class MyClass {
    constructor(name, weather) {
        this.name = name;
        this.weather = weather;
    }
    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log("Today is " + this.weather + ".");
    }
}
let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

JavaScript classes will be familiar if you have used another mainstream language such as Java or C#. The class keyword is used to declare a class, followed by the name of the class, which is MyClass in this case. The constructor function is invoked when a new object is created using the class, and it provides an opportunity to receive data values

and do any initial setup that the class requires. In the example, the constructor defines name and weather parameters that are used to create variables with the same names. Variables defined like this are known as properties.
Classes can have methods, which are defined as functions, albeit without needing to use the function keyword. There is one method in the example, called printMessages, and it uses the values of the name and weather properties to write messages to the browser's JavaScript console.
NOTA: Classes can also have static methods, denoted by the static keyword. Static methods belong to the class rather than the objects they create. (Example of a static method in Listing 6-14)

*The **new** keyword is used to create an object from a class, like this:*
```
...
let myData = new MyClass("Adam", "sunny");
...
```
This statement creates a new object using the `MyClass class` *as its template.* `MyClass` *is used as a function in this situation, and the arguments passed to it will be received by the constructor function defined by the class. The result of this expression is a new object that is assigned to a variable called* `myData`*.*
Once you have created an object, you can access its properties and methods through the variable to which it has been assigned, like this:
```
...
myData.printMessages();
...
```
This example produces the following results in the browser's JavaScript console:
```
Hello Adam.
Today is sunny.
```

6.4) Defining Class Getter and Setter Properties
JavaScript classes can define properties in their constructor, resulting in a variable that can be read and modified elsewhere in the application. Getters and setters appear as regular properties outside of the class, but they allow the introduction of additional logic, which is useful for validating or transforming new values or generating values programmatically, as shown in Listing 6-5.

```
class MyClass {
    constructor(name, weather) {
        this.name = name;
        this._weather = weather;
    }
    set weather(value) {
        this._weather = value;
    }
    get weather() {
        return `Today is ${this._weather}`;
    }
    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log(this.weather);
    }
}
let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

The getter and setter are implemented as functions preceded by the get or set keyword. There is no notion of access control in JavaScript classes, and the convention is to prefix the names of internal properties with an underscore (the _ character). In the listing, the weather property is implemented with a setter that updates a property called `_weather` *and with a getter that incorporates the* `_weather` *value in a template string. This example produces the following results in the browser's JavaScript console:*

```
Hello Adam.
Today is sunny
```

6.5) Class Inheritance: *Classes can inherit behavior from other classes using the extends keyword:*

```
class MyClass {
    constructor(name, weather) {
        this.name = name;
        this._weather = weather;
    }
    set weather(value) {
        this._weather = value;
    }
    get weather() {
        return `Today is ${this._weather}`;
    }
    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log(this.weather);
    }
}
class MySubClass extends MyClass {
    constructor(name, weather, city) {
        super(name, weather);
        this.city = city;
    }
    printMessages() {
        super.printMessages();
        console.log(`You are in ${this.city}`);
    }
}
let myData = new MySubClass("Adam", "sunny", "London");
myData.printMessages();
```

*The **extends** keyword is used to declare the class that will be inherited from, known as the superclass or base class. In the listing, MySubClass inherits from MyClass. The super keyword is used to invoke the superclass's constructor and methods. The MySubClass builds on the MyClass functionality to add support for a city, producing the following results in the browser's JavaScript console:*

```
Hello Adam.
Today is sunny
You are in London
```

6.6) JS Modules: *JavaScript modules are used to manage the dependencies in a web application, which means you don't need to manage a large set of individual code files to ensure that the browser downloads all the code for the application. Instead, during the compilation process, all of the JavaScript files that the application requires are combined into a larger file, known as a bundle, and it is this that is downloaded by the browser.*

6.6.1) *Creating and Using Modules -export*

*Each **TypeScript or JavaScript file that you add to a project is treated as a module**. To demonstrate, I created a folder called /modules in the /src folder, added to it a file called* `NameAndWeather.ts`*, and added the code shown in Listing 6-7.*

Listing 6-7. The Contents of the NameAndWeather.ts File in the src/modules Folder

```
export class Name {
    constructor(first, second) {
        this.first = first;
        this.second = second;
    }
    get nameMessage() {
        return `Hello ${this.first} ${this.second}`;
    }
}
export class WeatherLocation {
    constructor(weather, city) {
        this.weather = weather;
        this.city = city;
    }
    get weatherMessage() {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

The classes, functions, and variables defined in a JavaScript or TypeScript file can be accessed only within that file by default. The ***export*** *keyword is used to make features accessible outside of the file so that they can be used by other parts of the application. In the example, I have applied the* ***export*** *keyword to the Name and WeatherLocation classes, which means they are available to be used outside of the module.*
NOTA: *the convention in Angular applications is* ***to put each class into its own file****, which means that each class is defined in its own module and that you will see the* ***export*** *keyword in the listings.*

6.6.2) *Creating and Using Modules -import*

The ***import*** *keyword is used to declare a dependency on the features that a module provides. In Listing 6-8, I have used Name and WeatherLocation classes in the main.ts file, and that means I have to use the* ***import*** *keyword to declare a dependency on them and the module they come from.*

Listing 6-8. Importing Specific Types in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");

console.log(name.nameMessage);
console.log(loc.weatherMessage);
```

This is the way that I use the ***import*** *keyword in most of the examples in this book. The keyword is followed by curly braces that contain a comma-separated list of the features that the code in the current files depends on, followed by the* ***from*** *keyword, followed by the* ***module name****. In this case, I have imported the Name and WeatherLocation classes from the* **NameAndWeather.ts** *module in the modules folder. Notice that the file extension is not included when specifying the module.*

6.6.3) Duplicating Class Names *In complex projects that have lots of dependencies, it is possible that you will need to use two classes with the same name from different modules. To re-create this situation, I created a file called DuplicateName.ts in the src/modules folder and defined the class shown in Listing 6-9.*

Listing 6-9. The Contents of the DuplicateName.ts File in the src/modules Folder

```
export class Name {
    get message() {
        return "Other Name";
    }
}
```

This class doesn't do anything useful, but it is called Name, which means that importing it using the approach in Listing 6-8 will cause a conflict because the compiler won't be able to differentiate between the two classes with that name. The solution is to use the as keyword, which allows an alias to be created for a class when it is imported from a module, as shown in Listing 6-10.

Listing 6-10. Using a Module Alias in the main.ts File in the src Folder

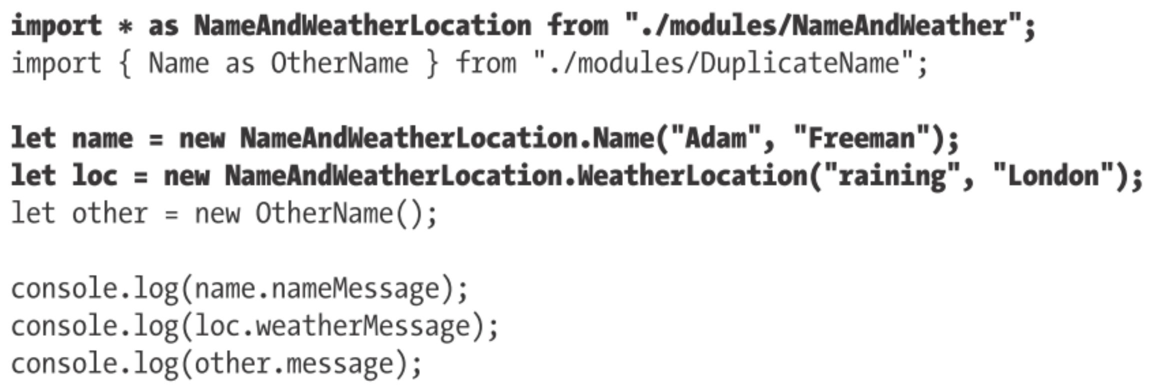
```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```

The Name class in the DuplicateName module is imported as OtherName, which allows it to be used without conflicting with the Name class in the NameAndWeather module. This example produces the following output:

```
Hello Adam Freeman
It is raining in London
Other Name
```

6.6.4) ***Alternative: Import all ítems from NameAndWeather.ts with *, and create new object NameAndWeatherLocation:***

Listing 6-11. Importing a Module as an Object in the main.ts File in the src Folder

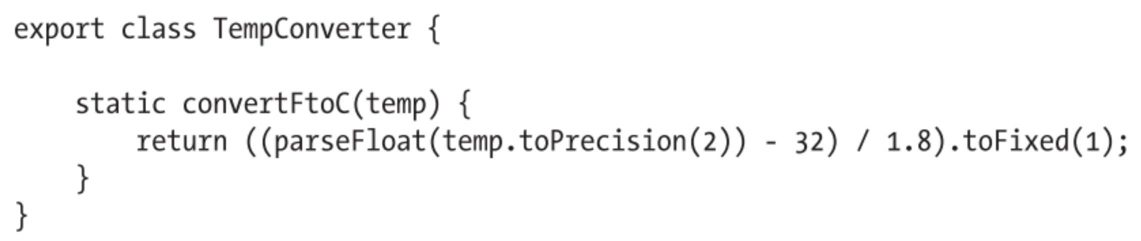


```
import * as NameAndWeatherLocation from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";

let name = new NameAndWeatherLocation.Name("Adam", "Freeman");
let loc = new NameAndWeatherLocation.WeatherLocation("raining", "London");
let other = new OtherName();

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```

```
(produces same output as Listing 6.10)
```

6.7) Enforcing Strict Type Annotations (main reason for using TS instead of JS)
6.7.1) *The classic Javascript is not type specific. This class Tempconverter has a function which expects a number in* `temp` *(on which the* **toPrecision** *method is called to set the number of floating-point digits), and returns a string. But no notion of this is directly communicated to the programmer:*

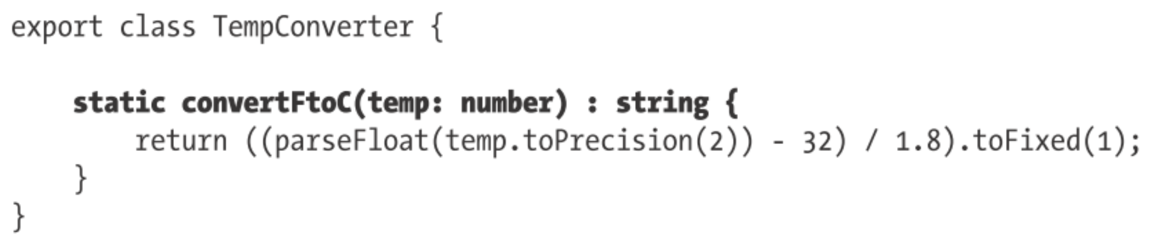
Listing 6-12. The Contents of the tempConverter.ts File in the src Folder



```
export class TempConverter {

    static convertFtoC(temp) {
        return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

With TS we can enforce both the argument and return value type as in any “normal” language:

Listing 6-14. Adding Type Annotations in the tempConverter.ts File in the src Folder



```
export class TempConverter {

    static convertFtoC(temp: number) : string {
        return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

Listing 6-15. Using a Number Argument in the main.ts File in the src Folder, using the convertFtoC() in listing 6.15

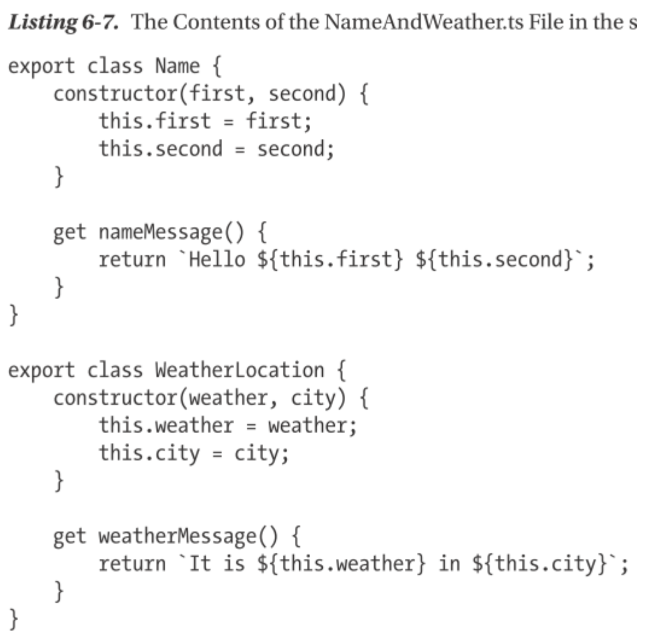
```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();
let cTemp = TempConverter.convertFtoC(38);

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
console.log(`The temp is ${cTemp}C`);
```

When you save the changes, you will see the following messages displayed in the browser's JavaScript console:

```
Hello Adam Freeman
It is raining in London
Other Name
The temp is 3.3C
```

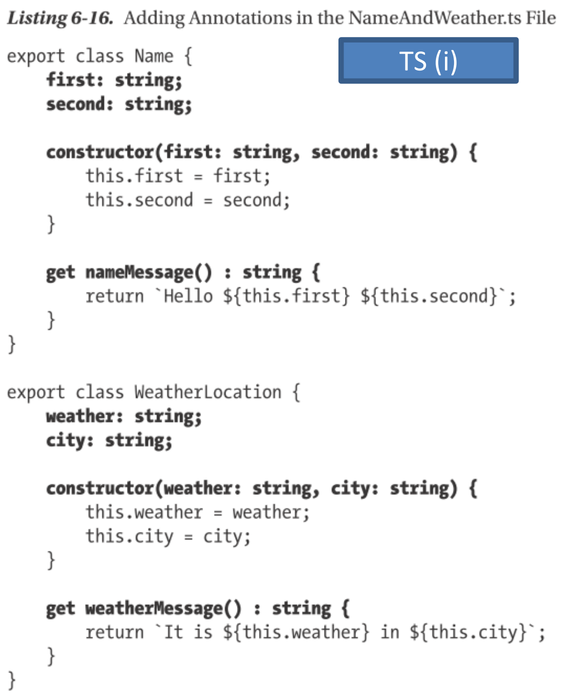
6.7.2) Adding strict typing in a complete class with TS:

JS



Listing 6-7. The Contents of the NameAndWeather.ts File in the s

```
export class Name {
    constructor(first, second) {
        this.first = first;
        this.second = second;
    }

    get nameMessage() {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
    constructor(weather, city) {
        this.weather = weather;
        this.city = city;
    }

    get weatherMessage() {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

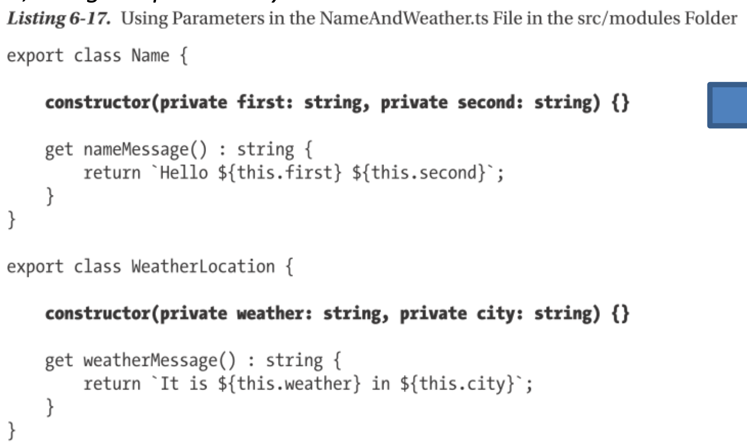


Listing 6-16. Adding Annotations in the NameAndWeather.ts File

TS (i)

```
export class Name {
    first: string;
    second: string;

    constructor(first: string, second: string) {
        this.first = first;
        this.second = second;
    }

    get nameMessage() : string {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
    weather: string;
    city: string;

    constructor(weather: string, city: string) {
        this.weather = weather;
        this.city = city;
    }

    get weatherMessage() : string {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

Listing 6.16 is much more clear once the convention is learned, and avoids programming mistakes.TS allows a more concise form as in listing 6.17, using the private keyword:



Listing 6-17. Using Parameters in the NameAndWeather.ts File in the src/modules Folder

TS (ii)

```
export class Name {

    constructor(private first: string, private second: string) {}

    get nameMessage() : string {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {

    constructor(private weather: string, private city: string) {}

    get weatherMessage() : string {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

NOTE on Access Modifiers in Typescript (not present in JS):

Table 6-2. *The TypeScript Access Modifier Keywords*

Keyword	Description
`public`	This keyword is used to denote a property or method that can be accessed anywhere. This is the default access protection if no keyword is used.
`private`	This keyword is used to denote a property or method that can be accessed only within the class that defines it.
`protected`	This keyword is used to denote a property or method that can be accessed only within the class that defines it or by classes that extend that class.

6.7.3) Allowing different types of arguments: *TypeScript allows multiple types to be specified, separated using a bar (the | character). This can be useful when a method can accept or return multiple types or when a variable can be assigned values of different types. Listing 6-18 modifies the convertFtoC method so that it will accept number or string values.*

Listing 6-18.

```
export class TempConverter {

    static convertFtoC(temp: number | string): string {
        let value: number = (<number>temp).toPrecision
            ? <number>temp : parseFloat(<string>temp);
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

The type declaration for the temp parameter has changes to **number | string***, which means that the method can accept either type. This is called a* **union type***. Within the method, a type assertion is used to work out which type has been received. This is a slightly awkward process, but the parameter value is cast to a number value to check whether there is a toPrecision method defined on the result, like this:*

```
...
(<number>temp).toPrecision
...
```

The angle brackets (the < and > characters) are to declare a ***type assertion****, which will attempt to convert an object to the specified type. Or using the* ***as*** *keyword (same effect, more clear):*

Listing 6-19. Using the as Keyword in the tempConverter.ts File in the src Folder

```
export class TempConverter {

    static convertFtoC(temp: number | string): string {
        let value: number = (temp as number).toPrecision
            ? temp as number : parseFloat(<string>temp);
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

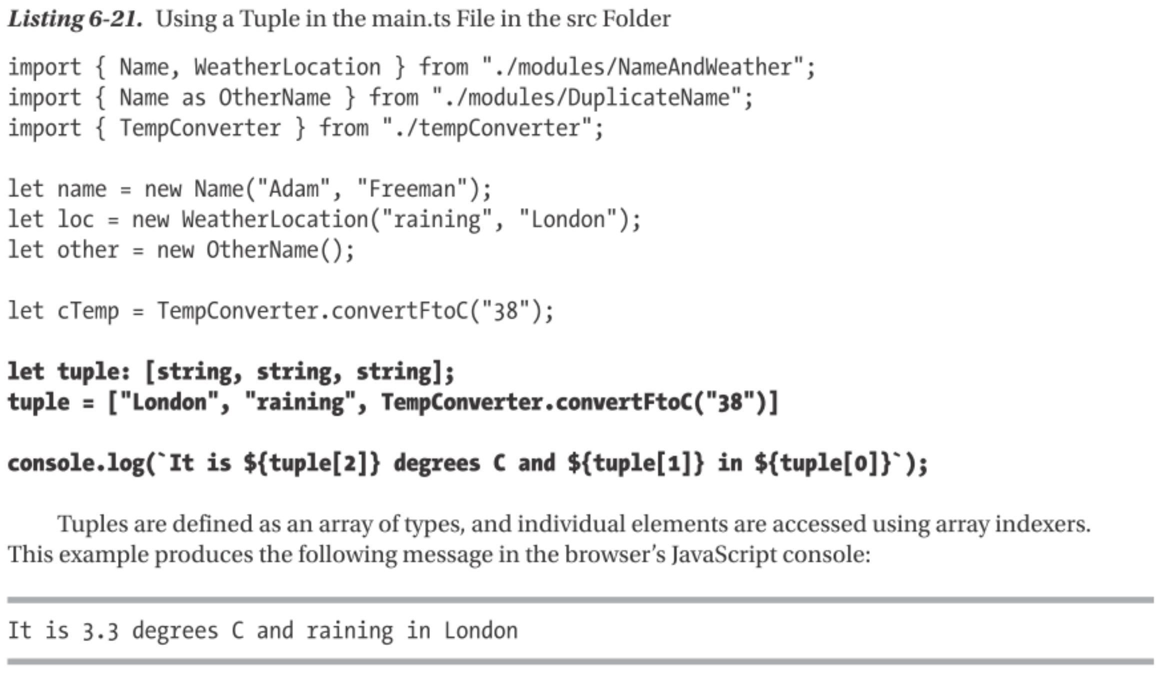
Same using a private access modifier, Listing 6-23 - tempConverter.ts:

```
export class TempConverter {
    static convertFtoC(temp: any): string {
        let value: number;
        if ((temp as number).toPrecision) {
            value = temp;
        } else if ((temp as string).indexOf) {
            value = parseFloat(<string>temp);
        } else {
            value = 0;
        }
        return TempConverter.performCalculation(value).toFixed(1);
    }
    private static performCalculation(value: number): number {
        return (parseFloat(value.toPrecision(2)) - 32) / 1.8;
    }
}
```

The `performCalculation()` *method is marked as private, which means the TypeScript compiler will report an error code if any other part of the application tries to invoke the method.*

6.8) Using Tuples

Tuples are fixed-length arrays, where each item in the array is of a specified type. This is a vague-sounding description because tuples are so flexible. As an example, Listing 6-21 uses a tuple to represent a city and its current weather and temperature.



Listing 6-21. Using a Tuple in the main.ts File in the src Folder

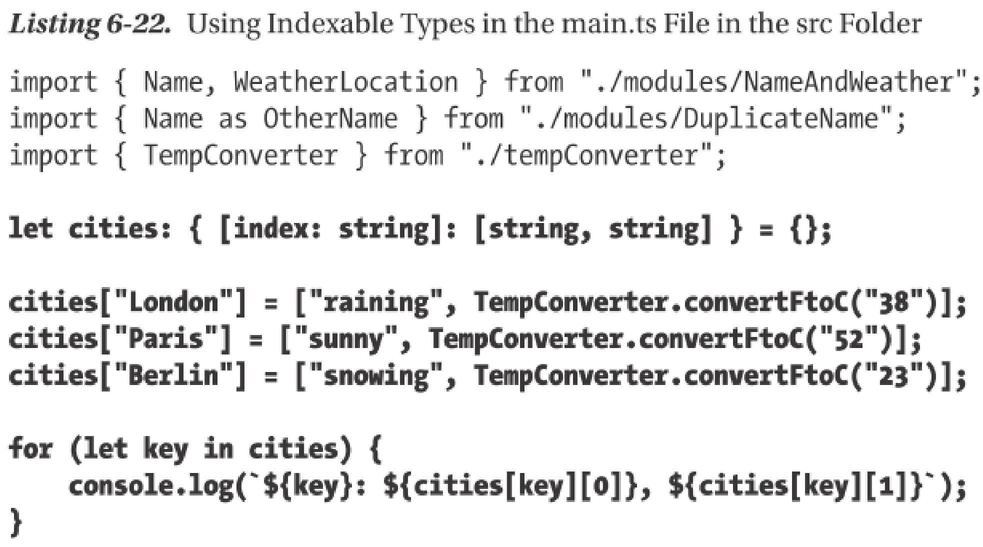
```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

let cTemp = TempConverter.convertFtoC("38");

let tuple: [string, string, string];
tuple = ["London", "raining", TempConverter.convertFtoC("38")]

console.log(`It is ${tuple[2]} degrees C and ${tuple[1]} in ${tuple[0]}`);
```

Tuples are defined as an array of types, and individual elements are accessed using array indexers. This example produces the following message in the browser's JavaScript console:

```
It is 3.3 degrees C and raining in London
```

6.8) Using Indexable Types

Indexable types *associate a key with a value, creating a map-like collection that can be used to gather related data items together. In Listing 6-22, I have used an indexable type to collect together information about multiple cities.*



Listing 6-22. Using Indexable Types in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let cities: { [index: string]: [string, string] } = {};

cities["London"] = ["raining", TempConverter.convertFtoC("38")];
cities["Paris"] = ["sunny", TempConverter.convertFtoC("52")];
cities["Berlin"] = ["snowing", TempConverter.convertFtoC("23")];

for (let key in cities) {
    console.log(`${key}: ${cities[key][0]}, ${cities[key][1]}`);
}
```

The `cities` *variable is defined as an indexable type, with the key as a* ***string*** *and the data value as a* ***[string, string] tuple****. Values are assigned and read using array-style indexers, such as* `cities["London"]`. *The collection of keys in an indexable type can be accessed using a* `for...in loop`, *as shown in the example, which produces the following output in the browser's JavaScript console:*

```
London: raining, 3.3
Paris: sunny, 11.1
Berlin: snowing, -5.0
```

Only number and string values can be used as the keys for indexable types, but this is a helpful feature.

PARTE II / Usando Seshadri + Freeman
Seshadri cubre injectors, pero no pipes. Freeman no cubre generate components, por ejemplo.

PARTE III / Ionic + Cordova
Libro: Build Mobile Apps with Ionic 4 and Firebase - Hybrid Mobile App Development 2ºEd 2018 de Fu Cheng
***Ionic*→** https://github.com/ionic-team/ionic-framework

PARTE IV / TP Mongo Para MONGO DB → libros:
a) The Little Mongo DB Schema Design Book (Christian Kvalheim -2015)
b) Un libro integrador que utiliza Mongo DB y el resto de las plataformas, es Getting MEAN WITH MONGO, EXPRESS, ANGULAR, AND NODE (2ªed) - SIMON HOLMES - CLIVE HARBER https://github.com/cliveharber/gettingMean-2

PARTE V / Practica (29.7.20 inicio):
Sobre Video Clase 1 – B.Ducca:
En VM de Probook /home/MIoT2020/DAppM/practica/TP1

1. **Open with VSC directory TP1, two terminals, in one:**
   ng generate tp1

   OK routing y, style CSS genera todo ok (Figura 1) – Aprendemos a capturar imágenes desde Linux con KAZAM, las guarda en **/home/MIoT2020/DAppM/practica/TP_Imgs/TP1**
2. ***Ojo con ng generate component → que sea "listado" con minúscula (como dice en el texto del TP), sinó lo manda al root.***