***Giáo trình***

**Lập trình Ứng dụng Di động**

**GS. Phan Viết Hoàng**

**Bài học 3 (Buổi 3)**

**Lập trình TypeScript**

# References

# ‘Mobile App Development with Ionic 2’, *Christopher Griffith, 2017*

# Lập trình Controller

# JavaScript ES6 and TypeScript

Over the past few years, web developers have seen an explosion in attempts to create “better” or more developer-centric versions of JavaScript. CoffeeScript, AtScript, Dart, ES6, TypeScript, and so on have sought to improve on standard JavaScript. Each of these languages sought to extend JavaScript by providing features and functionality aimed at modern application development. But each solution had to deal with the fact that our modern browsers use a version of JavaScript known formally as ECMAScript 5 (ES5), meaning that each solution would need to output its efforts into standard JavaScript.

In order to use either of these modern language options, our code will have to be *transpiled* into ES5-based JavaScript. If you have never heard of the term “*transpiling*” before, it is a process of taking the code written in one language and converting into another.

Currently, there are two primary choices if you want to use next-generation JavaScript: ES6 or TypeScript. ES6 is the next official version of JavaScript and was formally approved in June 2015, and over time it will be supported natively in our browsers. The other language option is TypeScript. Typescript is Microsoft’s extension of JavaScript that comes with powerful type-checking abilities and object-oriented features. It also leverages ES6 as part of its core foundation. TypeScript is a primary language for both Angular and Ionic application development.

Although none of our current browsers support either option, our code can be transpiled using tools like Babel or tsc. We don’t have to worry about setting up this system, as it is built into the default Ionic build process.

## Variables

With ES6, the way we can define our variables has improved. We can now specify a variable by using the keyword let .

In ES5, variables could only be defined using the keyword var, and they would be scoped to the nearest function. This was often problematic, as a variable could be accessible outside of the function that it was defined in.

for (var i = 0 ; i < 10; i++) {

console.log(i); //Output 0-9

}

console.log(i); // Outputs 10

The variable i is still available after the loop has finished, which is not quite the expected behavior.

By using the let keyword, this issue is no longer a problem, and the variable is scoped to its nearest block:

for (let i = 0 ; i < 10; i++) {

console.log(i); //Outputs 0-9

}

console.log(i); // Uncaught ReferenceError: i is not defined

Now, after the loop has executed, i is not known to the rest of the code. Whenever possible, use let to define your variables.

## Classes

JavaScript classes have been introduced in ES6 and are syntactical sugar over JavaScript’s existing prototype-based inheritance. The class syntax is *not* introducing a new object-oriented inheritance model to JavaScript. If you have developed using another object-oriented language like C# or Java, this new syntax should look familiar. Here is an example:

## class Rocket {

## landing(location) { }

## }

## class Falcon extends Rocket {

## constructor() {

## super();

## this.manufacturer = 'SpaceX';

## this.stages = 2;

## }

## landing(location) {

## if ((location == 'OCISLY') || (location == 'JRTI')){

## return 'On a barge';

## } else {

## return 'On land';

## }

## }

## }

## class Antares extends Rocket {

## constructor() {

## super();

## this.manufacturer = 'OrbitalATK';

## this.stages = 2;

## }

## landing(location) {

## console.log('In the ocean');

## }

## }

## Promises

The Promise object is used for deferred and asynchronous computations. A Promise represents an operation that hasn’t completed yet, but is expected in the future. This is exactly the type of functionality we need when interacting with remote servers or even loading local data. It provides a simpler method to handle asynchronous operations than traditional callback-based approaches.

A Promise can be in one of three states:

*Pending*

* The Promise’s outcome hasn’t yet been determined because the asynchronous operation that will produce its result hasn’t completed yet.

*Fulfilled*

* The asynchronous operation has completed, and the Promise has a value.

*Rejected*

* The asynchronous operation failed, and the Promise will never be fulfilled. In the rejected state, a Promise has a *reason* that indicates why the operation failed.

The primary API for a Promise is its then method, which registers callbacks to receive either the eventual value or the reason the Promise cannot be fulfilled.

Assuming we have a function sayHello that is asynchronous and needs to look up the current greeting from a web service based on the user’s geolocation, it may return a Promise:

var greetingPromise = sayHello();

greetingPromise.then(function (greeting) {

console.log(greeting); // 'Hello in the United States’

});

The advantage of this method is that while the function is awaiting the response from the server, the rest of our code can still function.

In case something goes wrong, like if the network goes down and the greeting can’t be fetched from the web service, you can register to handle the failure using the second argument to the Promise’s then method:

var greetingPromise = sayHello();

greetingPromise.then(

function (greeting) {

console.log(greeting); *// 'Hello in the United States’*

},

function (error) {

console.error('uh oh: ', error); *// 'Drat!’*

}

);

If sayHello succeeds, the greeting will be logged, but if it fails, then the reason (i.e., error) will be logged using console.error.

**Observables**

Many services with Angular use Observables instead of Promises. Observables are implemented through the use of the RxJS library. Unlike a Promise, which resolves to a single value asynchronously, an observable resolves to (or emits) multiple values asynchronously (over time).

In addition, Observables are cancellable and can be retried using one of the retry operators provided by the API, such as retry and retryWhen. Promises require the caller to have access to the original function that returned the Promise in order to have a retry capability.

**Template Strings**

One of the features of Angular is its built-in templating engine. In many cases, these templates are stored as external files. However, there are times when keeping them inline makes more sense. The difficulty has been writing long inline templates without having to resort to using concatenation or needing to escape any single or double quotes in the string.

ES6 now supports the use of backticks at the start and end of the string:

let template = ` <div> <h2>{{book.name}}</h2> <p> {{book.summary}} </p> </div> `;

Template strings do not have to remain static. You can perform string interpolation by using $(expression) placeholders:

let user = {name:'Rey'};

let template = ` <div>Hello, <span>${ user.name }</span></div> `;

**Template Expressions: ES6 versus Angular**

* ES6’s template expression are only meant for string replacement. If you need to evaluate a function or test a condition, use Angular template expressions instead.

## Arrow Functions

Arrow functions make our code more concise and simplify function scoping and the this keyword. By using arrow functions, we avoid having to type the function keyword, return keyword (it’s implicit in arrow functions), and curly brackets.

In ES5, we would have written a simple multiply function like this:

var multiply = function(x, y) {

return x \* y;

};

But in ES6, using the new arrow function formation, we can write the same function this way:

var multiply = (x, y) => {

return x \* y;

};

The arrow function example allows us to accomplish the same result with fewer lines of code and approximately half the typing.

One common use case for arrow functions is array manipulations. Take this simple array of objects:

var missions = [

{ name:'Mercury', flights:6 },

{ name:'Gemini', flights:10 },

{ name:'Apollo', flights:11 },

{ name:'ASTP', flights:1 },

{ name:'Skylab', flights:3 },

{ name:'Shuttle', flights:135 },

{ name:'Orion', flights: 0 }

];

We could create an array of objects with just the names or flights by doing this in ES5:

// ES5

console.log(missions.map(

function(mission) {

return mission.flights;

}

)); // [6, 10, 11, 1, 3, 135, 0]

Rewriting this using the arrow function, our code is more concise and easier to read:

## // ES6

## console.log(missions.map(

## mission=>mission.flights

## )); // [6, 10, 11, 1, 3, 135, 0]

## Types

TypeScript is a data-typed language that gives you compile-time checking. By default, TypeScript supports JavaScript primitives: string, number, and boolean:

let num: number;

let str: string;

let bool: boolean;

num = 123;

num = 123.456;

num = '123'; // Error

str = '123';

str = 123; // Error

bool = true;

bool = false;

bool = 'false'; // Error

TypeScript also supports typed arrays. The syntax is basically postfixing [] to any valid type annotation (e.g., boolean[]):

## let booleanArray: boolean[];

## booleanArray = [true, false];

## console.log(booleanArray[0]); // true console.log(booleanArray.length); // 2

## booleanArray[1] = true;

## booleanArray = [false, false];

## booleanArray[0] = 'false'; // Error!

## booleanArray = 'false'; // Error!

## booleanArray = [true, 'false']; // Error!

## Special Types

Beyond the primitive types, there are a few types that have special meaning in TypeScript. These are any, null, undefined, and void:

let someVar: any;

// Takes any and all variable types

someVar = '123';

someVar = 123;

The null and undefined JavaScript literals are effectively treated as the same as the any type:

## 

## var str: string;

## var num: number;

## // These literals can be assigned to anything

## str = undefined;

## num = null;

## Typing Functions

Not only can you type variables, but you can also type the results of a function call:

function sayHello(theName: string): string {

return 'Hello, '+theName;

}

If you try to call sayHello and assign the result to an incorrect data type, the compiler will throw an error.

## :void

Use :void to signify that a function does not have a return type:

# 

# function log(message): void {

# console.log(message);

# }

# Kết nối View

# Angular 2

Angular 2 is the next version of Google’s incredibly popular MV\* framework. This new version of Angular was announced at the ngEurope conference in October 2014. The Angular team revealed that this version of Angular would be a significant revision. In many ways, this new Angular was a completely new framework, sharing only its name and some notional references to the original version. This announcement certainly generated a lot of concern and uncertainty about the future of the framework. This was equally true with the Ionic community and within the Ionic team. What were the changes that would be required? How much relearning would existing Ionic developers need to undertake to continue to work with Angular?

But as the shock wore off, it became clearer that this evolution of the Angular framework was for the better. The framework was becoming faster, cleaner, more powerful, and also easier to understand. The Angular 2 team also took another bold gamble and looked at the web to come, not the web that was. So they decided to embrace many of the latest and emerging web standards and develop it in next generation of JavaScript.

At ngConf 2016, the Angular team announced that with release candidate 1, the framework will be simply known as Angular instead of Angular 2. For the purposes of this book, we will still refer to it as Angular 2.

So let’s take a look at some of these changes to Angular in more detail.

## Components

One of the biggest changes from Angular 1 to Angular 2 is the fact that we no longer rely on scope, controllers, or to some degree directives. The Angular team adopted a component-based approach to building elements and their associated logic. Those who have developed applications with more traditional frameworks are very familiar with this type of model. The fact is that we are developing on a platform originally designed to read physics papers and not to build applications upon.

Here is what a sample Angular component looks like:

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

@Component({

selector: 'my-first-component',

template:

` < div > Hello, my name is {{ name}}.

< button (click) =" sayMyName()" > Log my name </ button > </ div > `

})

export class MyComponent {

constructor() {

this.name = 'Inigo Montoya'

}

sayMyName() {

console.log(' Hello. My name is ' + this.name + '. You killed my father. Prepare to die.')

}

}

This is the exact same model that Ionic uses to generate its component library. In fact, there is nothing that prevents you from extending your Ionic application to use your own custom components.

Let’s look at this code snippet in greater detail.

First, the code imports the Component module from the Angular library. In Angular this is how dependency injection is handled. With Release Candidate 1 (RC1), the Angular team broke the library into smaller modules, as well as dropped the “2” suffix in the library.

Next, we use the @Component decorator to provide some metadata about our code to the compiler. We define the custom HTML selector to use. So when we use <my-first-component></my-first-component>, the associated template will be inserted into the DOM. Templates can come into two fashions: inline as shown here, or as an external reference. If you need to span your template across multiple lines for readability, make sure you use the backtick ( ` ) instead of a single quote ( ' ) to define the template string. We will look at templates in more detail later in this chapter.

After the decorator, we export the class definition itself, MyComponent. Within this constructor of this class, the code sets the name variable to "Inigo Montoya". Unlike Angular 1, and JavaScript in general, Angular 2 has a much tighter control over the scope of variables.

Finally, this sample class has a public method of sayMyName that will write a string to the console. As you work more with Ionic and Angular 2, this new method of creating components and pages will become more familiar and natural to you.

## Inputs

Since Angular 2 is built using a component model, it needs a mechanism to pass information into the component itself. This is handled via Angular’s Input module. Let’s look at a simple component, <current-user>, that will need to know about a user argument in order for it to perform its code. The actual markup would look like this:

< current-user [user] =" currentUser" > </ current-user >

while the component itself would look like this:

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

@Component({

selector: 'current-user',

template: '< div >{{ user.name}} </ div >'

})

export class UserProfile {

@Input() user;

constructor() {}

}

Within the class definition, there is now an @Input binding to the user variable. With this binding in place, Angular will pass in the currentUser variable into the component, thus enabling the template to render out the user.name value.

This is how Ionic’s components also function. We will pass in data and configuration parameters using the same system as this example.

## 

## Templates

Templates are HTML fragments that Angular combines with specific elements and attributes to generate the dynamic content. For the most part, the templating system in Angular 2 did not change that much.

**{ }: Rendering**

< div > Hello, my name is {{ name}}. </ div >

However, unlike Angular 1, this data binding is one way. By doing so, the number of event listeners that were generated have been reduced, and thus, performance improved.

### 

### [ ]: Binding properties

When a component needs to resolve and bind a variable, Angular now uses the [] syntax. We touched on this earlier in this chapter when covering Inputs.

If we have this.currentColor in our component, we would pass this variable into our component, and Angular would ensure that the values would stay updated:

### <card-header [themeColor]="currentColor"></card-header>

### ( ): Event handling

In Angular 1, we would use custom directives to listen for user events, like clicking an element (like ng-click). Angular 2 has taken a cleaner approach and just wraps the event you want to listen for in parentheses and then assigns that to a function in the component:

### <my-component (click)="onUserClick($event)"></my-component>

### [( )]: Two-way data binding

By default, Angular no longer establishes two-way data binding. If you do need to have this functionality, the new syntax is actually a shorthand notation of the binding property and the event-handling syntaxes:

<input [(ngModel)]="userName">

The this.userName value of your component will stay in sync with the input value.

### \*: The asterisk

The use of the asterisk before certain directives tells Angular to treat our template in a special fashion. Instead of rendering the template as is, it will apply the Angular directive to it first. For example, ngFor takes our <my-component> and stamps it out for each item in items, but it never renders our initial <my-component> since it’s a template:

## <my-component \*ngFor="let item of items"> </my-component>

## Events

Events in Angular 2 use the parentheses notation in templates and trigger methods in a component’s class. For example, assume we have this template:

<button (click)="clicked()">Click</button>

and this component class:

@Component(...) class MyComponent { clicked() { } }

Our clicked() method in the component will be called when the button is clicked.

In addition, events in Angular 2 behave like normal DOM events. They can bubble up and propagate down.

If we need access to the event object, simply pass in the $event as a parameter in the event callback function:

<button (click)="clicked($event)"></button>

and the component class would become:

@Component(...) class MyComponent { clicked( event) { } }

### Custom events

What if your component needs to broadcast a custom event to another component? Angular 2 makes this process quite easy.

In our component, we import the Output and EventEmitter modules. Then we define our new event, userUpdated, by using the @Output decorator. This event is an instance of an EventEmitter:

import {Component, Output, EventEmitter} from '@ angular/ core';

@Component({

selector: 'user-profile',

template: '< div > Hi, my name is </ div >'

})

export class UserProfile {

@Output() userDataUpdated = new EventEmitter();

constructor() {

// Update user // ... this.userDataUpdated.emit( this.user);

}

}

When we want to trigger the broadcast of the event, you simply call the emit method on the custom event type and include any parameters to be transmitted with the event.

Now when we used this component elsewhere in our app, we can bind the event that user-profile emits:

<user-profile (userDataUpdated)="userProfileUpdated($event)"></user-profile>

When we import our UserProfile component into our new component, it can now listen for the userProfileUpdated event that is broadcasted:

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

import {UserProfile} from './ user-profile';

export class SettingsPage {

constructor(){}

userProfileUpdated( user) {

// Handle the event

}

}

**Life cycle events**

Both the Angular app and its components offer life cycle hooks that give developers access to each of the critical steps as they occur. These events are usually related to their creation, their rendering, and their destruction.

### NgModule

The Angular team reworked the method of bootstrapping your application through the use of the NgModule function. This was done toward the end of the release candidate cycle for Angular, so it might come as a surprise to some. The @NgModule takes a metadata object that tells Angular how to compile and run module code. In addition, @NgModule allows you to declare all your dependencies up front, instead of having to declare them multiple times in an app:

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

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

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

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule {}

This code sample shows a basic app.module.ts file that will use the BrowserModule to enable the Angular app to properly run in a browser, then both declare and bootstrap the AppComponent.

This module is in turned used by the *main.ts* file to perform the actual bootstrapping:

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

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

const platform = platformBrowserDynamic();

platform.bootstrapModule( AppModule);

This sample code initializes the platform that your application runs in, then uses the platform to bootstrap your AppModule. The Ionic starter templates will generate the necessary modules for you.

Another benefit of this system is it enables us to use the Ahead of Time (AoT) compiler, which provides for much faster applications.

### Component init event

When a component is created, its constructor function is called. Within the constructor, any initialization we might need to perform on the component can occur. However, if our component is dependent on information or properties from a child component, we will not have access to that data.

Angular provides the ngOnInit event in order to handle this need to wait until the component initialization is truly complete. Our component can wait for this method to be triggered by the framework. Then all our properties are resolved and available to be used by the component.

### Component life cycle events

Beyond ngOnInit, there are several other life cycle events for a component:

ngOnDestroy

* This method is called before the component is destroyed by the framework. This would be where you unsubscribe observables and detach event handlers to avoid memory leaks.

ngDoCheck

* This method provides the ability to perform custom change detection.

ngOnChanges(changes)

* This method is called when one of the component’s bindings have changed during the checking cycle. The method’s parameter will be an object in the format:

{

'prop': PropertyUpdate

}

*ngAfterContentInit()*

* Unlike ngOnInit, which is called before the content has been rendered, this method is called once that content is first rendered on the view.

*ngAfterContentChecked*

* This method is called after Angular checks the bindings of the external content that it projected into its view.

*ngAfterViewInit*

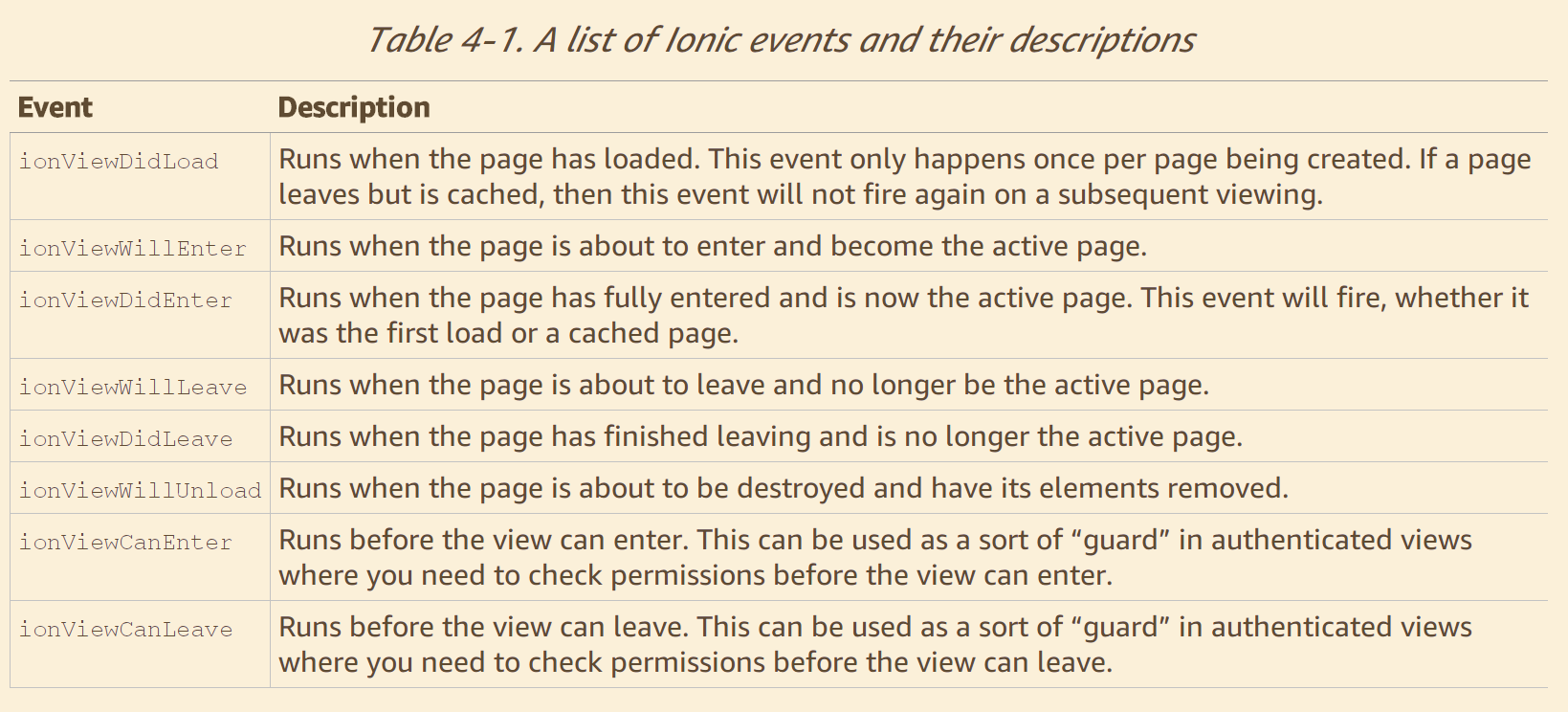
* After Angular creates the component’s view(s), this method is triggered.

*ngAfterViewChecked*

# The final method during the component initialization process, this will be call once all the data bindings are resolved in the component’s views.

# Ionic Events

Although you can use the Angular events outlined, it is recommended that you use the Ionic events instead. Table 4-1 lists a description of their triggers.



## Pipes

Pipes, previously known as “Filters,” transform a value into a new value, like localizing a string or converting a floating-point value into a currency representation:

< p > The author's birthday is {{ birthday | date }} </ p >

If the birthday variable is a standard JavaScript Date object, it will look like Thu Apr 18 1968 00: 00: 00 GMT-0700 (PDT). Certainly not the most human-readable format. However, within the interpolation expressed in our template, our birthday value is passed through the pipe operator ( | ) to the Date pipe function on the right, thus rendering the author’s birthday as April 18, 1968. Angular comes with a set of commonly used pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, and PercentPipe. They are all immediately available for use in any template.

**@ViewChild**

Often we need to read or write child component values or call a child’s component’s method. When the parent component class requires that kind of access, we inject the child component into the parent as a ViewChild:

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

import {UserProfile} from '../ user-profile';

@Component({

template: '< user-profile (click) =" update()" > </ user-profile >',

directives: [UserProfile]

})

export class MasterPage {

// we pass the Component we want to get

// assign to a public property on our class

// give it the type for our component

@ViewChild( UserProfile) userProfile: UserProfile

constructor() { }

update(){

this.userProfile.sendData();

}

}

Both the ViewChild module and UserProfile component are injected from the Angular Core. Within the Component decorator, we also must set the directives property to include a reference to our injected component. Our constructor contains our ViewChild decorator that set our userProfile variable to our injected component.

With those code elements in place, we are able to interact with our child component’s sendData method.