YASH Technologies

Angular 2 – ES6 & TypeScript

Topics to be covered

- Introduction to ES6
  - Transpillers
  - Hands on
  - Strict mode
  - Hoisting
  - Variable
  - Object
  - Class
- TypeScript
  - Introduction
  - Benefits
  - Installation
  - Compilation
  - Static typing
  - Interfaces
  - Class and OOPs - generics
  - Some useful references

ES6

ES6/ ECMASCRIT6/ ECMASCRIPT 2015

- Classes
- Constants
- Arrow function
- Generators

Let’s dive into ES6

ES6 - Transpillers

Problem if we use ES6:
All browsers are not updated to process the ES6 code.

Solution or Purpose of Transpillers:
A Transpiller takes the ES6 Source code and generates ES5 code that can run in every browser

There are two alternatives to transpile the ES6 code.

1. **Traceur** : It is a Google project

2. **BabelJS** : A Project started by young developer, Sebastian McKenzie (He was 17 Years old that time)
   https://babeljs.io/

Note : Angular 2 source code was transpiled first in Traceur before switching to TypeScript.
TypeScript is an open source language developed by Microsoft.

Create below code in any text editor (preferably in Visual Studio / ATOM)

```
var message="Hello World";
console.log(message);
```

1. Save it with test.js file name
2. Goto folder location where you saved this file.
3. Open folder in Command propmt
4. Run below command

```
node test.js
```

Note: node js should be installed on your system.

You will get the out as “Hello World”

Do a small change in your application, remore the var keyword, now save and rerun the node test.js command.

ES6 – Strict Mode

The fifth edition of ECMAScript specification introduces the String Mode.

```
"use strict"
message="Hello World";
console.log(message);
```

1. Save the changes and rerun the node test.js command.

Output

```
message="Hello World";
        ^
ReferenceError: message is not defined
```

You can also use the strict mode inside scope , like function or if block etc.it will make impose the strict mode inside the block.

ES6 – ES6 and Hoisting

The JavaScript engine, by default, moves declarations to the top. This feature is termed as **hoisting**. This feature applies to variables and functions. Hoisting allows JavaScript to use a component before it has been declared. However, the concept of hoisting does not apply to scripts that are run in the Strict Mode.

ES6 – Variable

Variable is a “named space in memory.” It works as a container to hold values in program.

```
//ES5 syntax for declaring a variable
var variable_name;
```

ES6 introduces following variable declaration syntax

- Using let
- Using const

ES6 – Variable – JavaScript and Dynamic Typeing

JavaScript is an un-typed language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically. This feature is termed as **dynamic typing**.

```
var message="Hello World";
message=10;
console.log(message);
```

While executing this code it will not show any error, rather it will assign new value type in message and message will be displayed.

ES6 – Variable – JavaScript variable scope

JavaScript defines only two scopes.

- Global Scope: a variable with global scope can be accessed from any part of the JS code
- Local Scope: a variable with local scope can be accessed within a function where it is declared.

```
var num=10;
function test(){
var num=100;
console.log("value of num in test()"+num);
}
console.log("value of num outside test()"+num);

test()
```

What will be the output of above code

ES6 – Variable – JavaScript variable scope


```
var a=10;
var a=20;
console.log(a);
```

What will be the output of above code

```
let a=10;
let a=20;
console.log(a);
```

What will be the output of above code

ES6 – Variable – JavaScript const


```
const a=10;
a=20;
console.log(a);
```

What will be the output of above code

Rules about constant:

- Constant cannot be reassigned a value
- Constant cannot be declared
- A Constant requires initialization
- A value assigned to constant value is mutable

ES6 – Variable – JavaScript Hoisting


```
function main(){
for(var i=1;i<=5;i++){
console.log(i);
}

console.log("value of i = "+i);
}

main();
```

What will be the output of above code

i is declared inside the for() but still available outside the for(), this is variable hoisting

ES6 – Objects

An **object** is an instance which contains a set of key value pairs. Unlike primitive data types, objects can represent multiple or complex values and can change over their life time. The values can be scalar values or functions or even array of other objects.

Object Declaration Syntax

```
var identifier = {
Key1:value,
Key2: function () {
//functions },

Key3: [“content1”,” content2”]
}
```

Syntax of accessing object’s property

```
objectName.property
```

ES6 – Objects

Write the below code in text editor, and execute it

```
var employee={
name: "pankaj sharma",
salary: 90000,
showDetail: function(){
console.log("Name : "+this.name);
console.log("Salary : "+this.salary);
}
}

employee.showDetail();
```

Here employee is an object having name, salary and showDetail as properties.

ES6 – Objects-constructor

JavaScript provides a special constructor function called **Object()** to build the object. The new operator is used to create an instance of an object. To create an object, the new operator is followed by the constructor method.

What will be the output of below code?

```
var testObj=new Object();
console.log(testObj);
```

Output : {}

Now if you want to add some property along with value to be associated. Then we use the below methods.

```
testObj.property=value;
Or
testObj[“key”]=value
```

ES6 – Objects-constructor

Check for below code:

```
var car=new Object();
car.make="Maruti";
car.model="Maruti Alto LX";
car.year=2008;
console.log(car);
```

ES6 – Objects-constructor

One more way of creating object: using function

```
function Car(){
this.make="Maruti"
this.model="Maruti Alto LX"
this.year=2008
}

var car=new Car();
console.log(car);
```

```
function Car(){
this.make="Maruti"
}

var car=new Car();
car.model="Maruti Alto Lx";
console.log(car);
```

ES6 – Objects (Creating Object using Object.create() method)

Objects can also be created using the **Object.create()** method. It allows you to create the prototype for the object you want, without having to define a constructor function.

```
var roles={
type:"Admin", // default role
showType: function(){
// show the type of role
console.log(this.type);
}
}

//create new super_role
var super_role=Object.create(roles);
super_role.showType();
```

Make changes to above code to create new roles as “Manager”

ES6 – Objects (Creating Object using Object.create() method)

Solution

```
var roles={
type:"Admin", // default role
showType: function(){
// show the type of role
console.log(this.type);
}
}

//create new super_role
var super_role=Object.create(roles);
super_role.showType();

//Create new manager_role
var manager_role=Object.create(roles);
manager_role.type="Manager";
manager_role.showType();
```

ES6 – Objects (Object.assign() function)

The **Object.assign()** method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Syntax

Object.assign(target,…sources);

```
var employee=new Object();
employee.name="Pankaj";
employee.salary=90000;

var employeeCopy=Object.assign({},employee);
console.log("employee: ",employee);
console.log("employeeCopy: ",employeeCopy);
employeeCopy.id=1004482;
console.log("employeeCopy: ",employeeCopy);
console.log("employee: ",employee);
```

ES6 – Objects (Object.assign() function)

The **Object.assign()** method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Syntax

Object.assign(target,…sources);

Object cloning

```
var employee=new Object();
employee.name="Pankaj";
employee.salary=90000;

var employeeCopy=Object.assign({},employee);
console.log("employee: ",employee);
console.log("employeeCopy: ",employeeCopy);
employeeCopy.id=1004482;
console.log("employeeCopy: ",employeeCopy);
console.log("employee: ",employee);
```

ES6 – Class

Write below code:

```
class Employee{
constructor(name,salary){
this.name=name;
this.salary=salary;
}

showEmployeeDetail(){
console.log('name : ',this.name);
console.log('salary : ',this.salary);
}
}

var employee=new Employee('pankaj',90000);
employee.showEmployeeDetail();
```

ES6 – Objects (Object.assign() function)

Object merging

```
var employee=new Object();
employee.name="Pankaj";
employee.salary=90000;

var employeeCopy=Object.assign({},employee);
console.log("employee: ",employee);
console.log("employeeCopy: ",employeeCopy);
employeeCopy.id=1004482;
console.log("employeeCopy: ",employeeCopy);
console.log("employee: ",employee);
```

TypeScript - Inroduction

TypeScript, a compile-to-JavaScript language designed for developers who build large and complex apps. It inherits many programming concepts from languages such as C# and Java.

Benefits of TypeScript

- Due to the static typing, code written in TypeScript is more predictable, and is generally easier to debug.
- Makes it easier to organize the code base for very large and complicated apps
- TypeScript has a compilation step to JavaScript that catches all kinds of errors before they reach runtime and break something.
- Angular 2 framework is written in TypeScript and it’s recommended that developers use the language in their projects as well.

TypeScript - Installation

Note : You need node.js and npm for experimenting with typescript.

The easiest way to setup TypeScript is via npm. Using the command below we can install the TypeScript package globally, making the TS compiler available in all of our projects:

```
npm install -g typescript
```

Open the command prompt and type below command. If everything goes well. It will display the typescript version installed on your system.

```
tsc -v
```

TypeScript - Editor

Any text editor is fine.

Preferred one:

- Microsoft Visual Studio
- ATOM
- Bracket

TypeScript - Compilation

Typescript is written with .ts extension. This code can not be run directly on browser, and need to be compiled in .js file. Typescript compiler will do this. There are three ways to compile the typescript.

1. Manually by using tsc command.
2. Using tool
3. Using automated task runner like *gulp*

Preferred one:

- Manually by using tsc command

```
tsc test.ts
```

```
tsc test.ts main.ts
```

```
tsc *.ts
```

We can also on the –watch option with compilation, that will enable us for the changes in .ts file

```
tsc test.ts --watch
```

tsconfig.json file holds the setting for automatic compilation of all .ts files.

TypeScript – Static Typing

A very distinctive feature of TypeScript is the support of static typing. This means that you can declare the types of variables, and the compiler will make sure that they aren't assigned the wrong types of values. If type declarations are omitted, they will be inferred automatically from your code.

Here is an example. Any variable, function argument or return value can have its type defined on initialization:

```
var burger:string='hamburger', // string
calories:number=300, // numeric
tasty : boolean=true; // boolean

function placeOrder(food: string, energy: number):void{
console.log("Our "+food +" has "+ energy +" calories");
}

placeOrder(burger,calories);
```

TypeScript – Static Typing

Try for writing below code in .ts file

```
var testy:boolean='I have not tried it!';
```

Compile it using tsc test.tx command.
And see what happens.
Check for test.js file for any changes.
Same will be applied on function calling as well.

TypeScript – Commonly used data types

Number
String
Boolean
Any : A variable with this type can have it's value set to string, number, or anything else
Arrays: Has two possible syntax: my_arr: number[] or my_arr:Array<number>
Void: used on function that do not return any value
To know more in detail
http://www.typescriptlang.org/docs/handbook/basic-types.html

TypeScript – Interfaces

Interfaces are used to type-check whether an object fits a certain structure. By defining an interface we can name a specific combination of variables, making sure that they will always go together. When translated to JavaScript, interfaces disappear – their only purpose is to help in the development stage.

```
// Here we define our Food interface, its properties, and their types.
interface Food {
name: string;
calories: number;
}

// We tell our function to expect an object that fulfills the Food interface.
// This way we know that the properties we need will always be available.
function placeOrder(food: Food): void{
console.log("Our " + food.name + " has " + food.calories + " calories.");
}

// We define an object that has all of the properties the Food interface expects.
// Notice that types will be inferred automatically.
var ice_cream = {
name: "ice cream",
calories: 200
}

placeOrder(ice_cream);
```

TypeScript – Classes and OOPs

When building large scale apps, the object oriented style of programming is preferred by many developers, most notably in languages such as Java or C#. TypeScript offers a class system that is very similar to the one in these languages, including inheritance, abstract classes, interface implementations, setters/getters, and more.

Since the most recent JavaScript update (ECMAScript 2015), classes are native to vanilla JS and can be used without TypeScript. The two implementation are very similar but have their differences, TypeScript being a bit more strict.

TypeScript – Classes and OOPs


```
class Menu{
// Our properties:
// By default they are public, but can also be private or protected.
items: Array<string>; // The items in the menu, an array of strings.
pages: number; // How many pages will the menu be, a number.

// A straightforward constructor.
constructor(item_list: Array<string>, total_pages: number) {
// The this keyword is mandatory.
this.items = item_list;
this.pages = total_pages;
}

// Methods
list(): void {
console.log("Our menu for today:");
for(var i=0; i<this.items.length; i++) {
console.log(this.items[i]);
}
}

}

// Create a new instance of the Menu class.
var sundayMenu = new Menu(["pancakes","waffles","orange juice"], 1);

// Call the list method.
sundayMenu.list();
```

TypeScript – Classes and OOPs (Inheritance)


```
class HappyMeal extends Menu{
// Properties are inherited

// A new constructor has to be defined.
constructor(item_list: Array<string>, total_pages: number) {
// In this case we want the exact same constructor as the parent class (Menu),
// To automatically copy it we can call super() - a reference to the parent's constructor.
super(item_list, total_pages);
}

// Just like the properties, methods are inherited from the parent.
// However, we want to override the list() function so we redefine it.
list(): void{
console.log("Our special menu for children:");
for(var i=0; i<this.items.length; i++) {
console.log(this.items[i]);
}

}
}
// Create a new instance of the HappyMeal class.
var menu_for_children = new HappyMeal(["candy","drink","toy"], 1);

// This time the log message will begin with the special introduction.
menu_for_children.list();
```

TypeScript – Generics

Generics are templates that allow the same function to accept arguments of various different types. Creating reusable components using generics is better than using the any data type, as generics preserve the types of the variables that go in and out of them.

A quick example would be a script that receives an argument and returns an array containing that same argument.

TypeScript – Generics


```
// The <T> after the function name symbolizes that it's a generic function.
// When we call the function, every instance of T will be replaced with the
actual provided type.

// Receives one argument of type T,
// Returns an array of type T.

function genericFunc<T>(argument: T): T[] {
var arrayOfT: T[] = []; // Create empty array of type T.
arrayOfT.push(argument); // Push, now arrayOfT = [argument].
return arrayOfT;
}

var arrayFromString = genericFunc<string>("beep");
console.log(arrayFromString[0]); // "beep"
console.log(typeof arrayFromString[0]) // String

var arrayFromNumber = genericFunc(42);
console.log(arrayFromNumber[0]); // 42
console.log(typeof arrayFromNumber[0]) // number
```

TypeScript – Generics

The first time we called the function we manually set the type to string. This isn't required as the compiler can see what argument has been passed and automatically decide what type suits it best, like in the second call. Although it's not mandatory, providing the type every time is considered good practice as the compiler might fail to guess the right type in more complex scenarios.

http://www.typescriptlang.org/docs/handbook/generics.html

TypeScript – Module

Another important concept when working on large apps is modularity. Having your code split into many small reusable components helps your project stay organized and understandable, compared to having a single 10000-line file for everything.

TypeScript introduces a syntax for exporting and importing modules, but cannot handle the actual wiring between files. To enable external modules TS relies on third-party libraries: require.js for browser apps and CommonJS for Node.js. Let's take a look at a simple example of TypeScript modules with require.js:

We will have two files. One exports a function, the other imports and calls it.

TypeScript – Module

Another important concept when working on large apps is modularity. Having your code split into many small reusable components helps your project stay organized and understandable, compared to having a single 10000-line file for everything.

TypeScript introduces a syntax for exporting and importing modules, but cannot handle the actual wiring between files. To enable external modules TS relies on third-party libraries: require.js for browser apps and CommonJS for Node.js. Let's take a look at a simple example of TypeScript modules with require.js:

http://www.typescriptlang.org/docs/handbook/modules.html
http://requirejs.org/docs/download.html#requirejs

TypeScript – Further Reading

Namespaces:
http://www.typescriptlang.org/docs/handbook/namespaces.html

Enum
http://www.typescriptlang.org/docs/handbook/enums.html

Advanced Types
http://www.typescriptlang.org/docs/handbook/advanced-types.html

JSX
http://www.typescriptlang.org/docs/handbook/jsx.html

tsConfig.json
https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Thank You!

Email: info@yash.com
Web: www.yash.com