

**Angular** is a platform for building mobile and web applications. Angular builds applications using HTML and TypeScript.

HTML you all might be already familiar with.

What is Typescript? Angular is written in TypeScript.

AngularJS is a JavaScript-based open-source front-end web framework.

**ECMAScript** is the scripting language that forms the basis of JavaScript.

Before we start building our application using Angular, let's first understand about ECMAScript and TypeScript.

### ECMAScript

ECMAScript (in short) referred as ES is a [general-purpose programming language](#). It is commonly used for [client-side scripting](#).

ECMAScript was created to standardize JavaScript, and **ES6** is the 6th version of ECMAScript, it was published in 2015.

### Working with “let” and “const”

ES6 introduced two important new JavaScript keywords: **let** and **const**.

Before ES6 the older versions of JavaScript had only two types of scope: **Global Scope** and **Function Scope**. These two keywords provide **Block Scope** variables (and constants) in JavaScript.

**Global** variables can be accessed from anywhere in a JavaScript program.

Variables declared **Locally** (inside a function) have **Function Scope**. **Local** variables can only be accessed from inside the function where they are declared.

Variables declared with the **var** keyword cannot have **Block Scope**.

Variables declared inside a block **{}** can be accessed from outside the block.

The **let** keyword allows you to declare a variable with block scope.

```
var x = 10;  
// Here x is 10  
{  
  let x = 2;
```

```
// Here x is 2
}  
// Here x is 10
```

Variables defined with **const** behave like **let** variables, except they cannot be reassigned:

```
const PI = 3.141592653589793;  
PI = 3.14;    // This will give an error  
PI = PI + 10; // This will also give an error
```

Declaring a variable with **const** is like **let** when it comes to **Block Scope**.  
JavaScript **const** variables must be assigned a value when they are declared.

You can change the properties of a constant object, but you can NOT reassign a constant object.

```
// You can create a const object:  
const car = {type:"Fiat", model:"500", color:"white"};
```

```
// You can change a property:  
car.color = "red";
```

```
// You can add a property:  
car.owner = "Johnson";
```

```
const car = {type:"Fiat", model:"500", color:"white"};  
car = {type:"Volvo", model:"EX60", color:"red"}; // ERROR
```

You can change the elements of a constant array, but you can NOT reassign a constant array.

```
// You can create a constant array:  
const cars = ["Saab", "Volvo", "BMW"];
```

```
// You can change an element:  
cars[0] = "Toyota";
```

```
// You can add an element:  
cars.push("Audi");
```

```
const cars = ["Saab", "Volvo", "BMW"];  
cars = ["Toyota", "Volvo", "Audi"]; // ERROR
```

## Working with ES6 Functions

A function is the set of input statements, which performs specific computations and produces output. It is a block of code that is designed for performing a particular task. It is executed when it gets invoked (or called). Functions allow us to reuse and write the code in an organized way. [Functions in JavaScript](#) are used to perform operations.

In [JavaScript](#), functions are defined by using function keyword followed by a **name** and **parentheses ()**. The function name may include digits, letters, dollar sign, and underscore. The brackets in the function name may consist of the name of parameters separated by commas. The body of the function should be placed within **curly braces {}**.

The syntax for defining a standard function is as follows:

```
function function_name() {  
    //body of the function  
}
```

To force the execution of the function, we must have to invoke (or call) the function. It is known as function invocation. The syntax for invoking a function is as follows:

```
function_name()
```

### **Example**

```
function hello() //defining a function  
{  
    console.log("hello function called");  
}  
hello(); //calling of function
```

In the above code, we have defined a function **hello()**. The pair of parentheses **{}** define the body of the function, which is called a scope of function.

### **Output**

```
hello function called
```

## Parameterized functions

Parameters are the names that are listed in the definition of the function. They are the mechanism of passing the values to functions.

The values of the parameters are passed to the function during invocation. Unless it is specified explicitly, the number of values passed to a function must match with the defined number of parameters.

### Syntax

The syntax for defining a parameterized function is:

```
function function_name( parameter1,parameter2 ,....parameterN) {  
    //body of the function  
}
```

### Example

In this example of parameterized function, we are defining a function **mul()**, which accepts two parameters **x** and **y**, and it returns their multiplication in the result. The parameter values are passed to the function during invocation.

```
function mul( x , y) {  
    var c = x * y;  
    console.log("x = "+x);  
    console.log("y = "+y);  
    console.log("x * y = "+c);  
}  
mul(20,30);
```

### Output

```
x = 20  
y = 30  
x * y = 600
```

## Default Function Parameters

In ES6, the function allows the initialization of parameters with default values if the parameter is undefined or no value is passed to it.

You can see the illustration for the same in the following code:

### For example

```
function show (num1, num2=200)
{
  console.log("num1 = " +num1);
  console.log("num2 = " +num2);
}
show(100);
```

In the above function, the value of **num2** is set to **200** by default. The function will always consider **200** as the value of **num2** if no value of **num2** is explicitly passed.

### Output

```
100 200
```

The default value of the parameter '**num2**' will get overwritten if the function passes its value explicitly. You can understand it by using the following example:

### For example

```
function show(num1, num2=200)
{
  console.log("num1 = " +num1);
  console.log("num2 = " +num2);
}
show(100,500);
```

### Output

```
100 500
```

## Understanding Spread Operator

### Syntax:

**var variablename1 = [...value];**

In the above syntax, ... is spread operator which will target all values in particular variable. When ... occurs in function call or alike, its called a spread operator. Spread operator can be used in many cases, like when we want to *expand, copy, concat, with math object*. Let's look at each of them one by one:

### Concat()

The concat() method provided by javascript helps in concatenation of two or more strings(String concat() ) or is used to merge two or more arrays. In case of arrays, this method does not change the existing arrays but instead returns a new array.

```
// normal array concat() method
let arr = [1,2,3];
let arr2 = [4,5];
arr = arr.concat(arr2);
console.log(arr); // [ 1, 2, 3, 4, 5 ]
```

Observe the Output

We can achieve the same output with the help of the spread operator, the code will look something like this:

```
// spread operator doing the concat job
let arr = [1,2,3];
let arr2 = [4,5];
arr = [...arr,...arr2];
console.log(arr); // [ 1, 2, 3, 4, 5 ]
```

### Math

The Math object in javascript has different properties that we can make use of to do what we want like finding the minimum from a list of numbers, finding maximum etc. Consider the case that we want to find the minimum from a list of numbers, we will write something like this:

```
console.log(Math.min(1,2,3,-1)); //-1
```

Now consider that we have an array instead of a list, this above `Math` object method won't work and will return `NaN`, like:

```
// min in an array using Math.min()

let arr = [1,2,3,-1];

console.log(Math.min(arr)); //NaN
```

When `...arr` is used in the function call, it “expands” an iterable object `arr` into the list of arguments

In order to avoid this `NaN` output, we make use of spread operator, like:

```
// with spread
let arr = [1,2,3,-1];

console.log(Math.min(...arr)); //-1
```

### Example of spread operator with objects

ES6 has added **spread** property to object literals in javascript. The spread operator (`...`) with objects is used to create copies of existing objects with new or updated values or to make a copy of an object with more properties. Let's take an example of how to use the spread operator on an object,

```
const user1 = {
  name: 'Jen',
  age: 22
};

const clonedUser = { ...user1 };
console.log(clonedUser);
```

Here we are spreading the **user1** object. All key-value pairs of the `user1` object are copied into the `clonedUser` object. Let's look on another example of merging two objects using the spread operator,

```
const user1 = {
  name: 'Jen',
  age: 22,
};

const user2 = {
  name: "Andrew",
```

```
    location: "Philadelphia"
  };
  const mergedUsers = {...user1, ...user2};
  console.log(mergedUsers)
```

**mergedUsers** is a copy of **user1** and **user2**. Actually, every enumerable property on the objects will be copied to mergedUsers object

## Understanding REST parameter

There are many built-in functions in JavaScript like `max` and `assign` that support an arbitrary number of arguments.

For instance:

- `Math.max(arg1, arg2, ..., argN)` – returns the greatest of the arguments.
- `Object.assign(dest, src1, ..., srcN)` – copies properties from `src1..N` into `dest`.

**Rest Parameters:** A function can be called with any number of arguments, no matter how it is defined.

If you consider the below example,

```
function sum(a, b) {
  return a + b;
}
```

```
alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of “excessive” arguments. But of course in the result only the first two will be counted.

The rest of the parameters can be included in the function definition by using three dots `...` followed by the name of the array that will contain them. The dots literally mean “gather the remaining parameters into an array”.

For instance, to gather all arguments into array `args`:

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}
```



```
alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6
```

We can choose to get the first parameters as variables and gather only the rest.

Here the first two arguments go into variables and the rest go into `titles` array:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar

  // the rest go into titles array
  // i.e. titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}
```

```
showName("Julius", "Caesar", "Consul", "Imperator");
```

### The rest parameters must be at the end

The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

```
function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
  // error

}
```

The `...rest` must always be last.

## Working with Classes in JavaScript

A **class** in terms of OOP is a blueprint for creating objects. A **class** encapsulates data for the object.

Does ES6 JavaScript have classes?

In **ES6** we **can** create **classes**. ... The **class** function basically creates a template that we **can** use to create objects later.

Objects in programming languages provide us with an easy way to model data. Let's say we have an object called *user*. The *user* object has **properties**: values that contain data about the user, and **methods**: functions that define actions that the user can perform.

The constructor() method **is** a special method called when an instance of the User **class** **is** created.



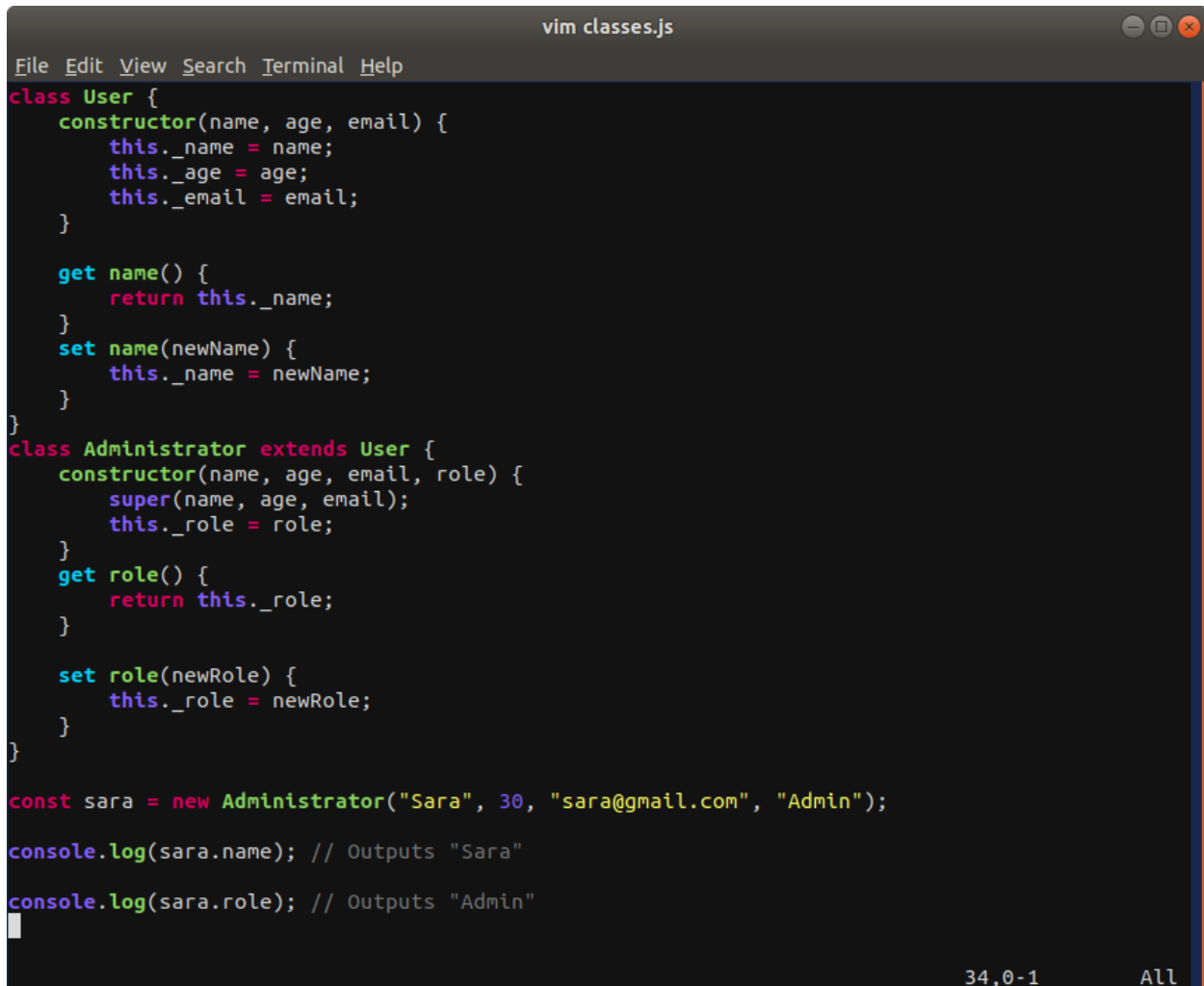
```
vim classes.js
File Edit View Search Terminal Help
class User {
  constructor(name, age, email) {
    this.name = name;
    this.age = age;
    this.email = email;
  }
  increaseAge() {
    this.age += 1;
  }
  static staticMethod() {
    console.log('I am a static method');
  }
}

const jeff = new User("Jeff", 30, "jeff@gmail.com");

jeff.increaseAge();

jeff.staticMethod(); // returns TypeError since staticMethod is not a method of jeff
User.staticMethod(); // outputs "I am a static method"
~
22,0-1 All
```

**Inheritance:** Classes can also inherit from other classes. The class being inherited from is called the **parent**, and the class inheriting from the parent is called the **child**. In our example, another class, let's say *Administrator*, can inherit the properties and methods of the *User* class:

A screenshot of a vim editor window titled 'vim classes.js'. The editor shows a JavaScript file with two classes: 'User' and 'Administrator'. The 'User' class has a constructor with parameters 'name', 'age', and 'email', and methods 'get name()' and 'set name(newName)'. The 'Administrator' class extends 'User' and has a constructor with parameters 'name', 'age', 'email', and 'role', and methods 'get role()' and 'set role(newRole)'. The 'Administrator' constructor calls 'super(name, age, email)' to inherit from 'User'. Below the class definitions, there is a constant 'sara' created as a new 'Administrator' object with values 'Sara', 30, 'sara@gmail.com', and 'Admin'. Two console log statements are present: 'console.log(sara.name);' and 'console.log(sara.role);'. The status bar at the bottom right shows '34,0-1' and 'All'.

34,0-1 All

In the above example, *User* is the parent and *Administrator* is the child. There's a few important things to note. Firstly, when we create the child class we need to state that it **extends** the parent class. Then we need to pass whatever properties we want to inherit from the parent to the child's constructor, as well as any new properties that we will define in the child class. Next, we call the **super** method. Notice that we pass it the values that we pass the child class when creating the *sara* object. These values are defined in the parent's constructor so we need to run it in order for the values to be instantiated. Now we can define our child class's properties and methods.

## Working with ES6 Exports and Imports

we can create modules in JavaScript. In a module, there can be classes, functions, variables, and objects as well. To make all these available in another file, we can use **export** and **import**. The **export** and **import** are the keywords used for exporting and importing one or more members in a module.

**Export:** You can export a variable using the **export** keyword in front of that variable declaration. You can also export a function and a class by doing the same.

- **Syntax for variable:**

```
export let variable_name;
```

- **Syntax for function:**

```
export function function_name() {  
    // Statements  
}
```

- **Syntax for class:**

```
export class Class_Name {  
    constructor() {  
        // Statements  
    }  
}
```

- **Example 1:** Create a file named **export.js** and write the below code in that file.

```
export let num_set = [1, 2, 3, 4, 5];  
  
export default function hello() {  
    console.log("Hello World!");  
}  
  
export class Greeting {  
    constructor(name) {  
        this.greeting = "Hello, " + name;  
    }  
}
```

- **Example 2:** In this example, we export by specifying the members of the module at the end of the file. We can also use alias while exporting using the **as** keyword.

```
let num_set = [1, 2, 3, 4, 5];
```

```

export default function hello() {
  console.log("Hello World!");
}

class Greeting {
  constructor(name) {
    this.greeting = "Hello, " + name;
  }
}

export { num_set, Greeting as Greet };

```

- **Note:** A default export should be specified here.

You can export members one by one. What's not exported won't be available directly outside the module:

```

export const myNumbers = [1, 2, 3, 4];
const animals = ['Panda', 'Bear', 'Eagle']; // Not available directly outside the module

export function myLogger() {
  console.log(myNumbers, animals);
}

export class Alligator {
  constructor() {
    // ...
  }
}

```

Or you can export desired members in a single statement at the end of the module:

```

export { myNumbers, myLogger, Alligator };

```

## Exporting with alias

You can also give an aliases to exported members with the `as` keyword:

```

export { myNumbers, myLogger as Logger, Alligator }

```

## Import:

Importing is also very straightforward, with the import keyword, members to be imported in curly brackets and then the location of the module relative to the current file:

You can import a variable using **import** keyword. You can specify one of all the members that you want to import from a JavaScript file.

- **Syntax:**

```
import member_to_import from "path_to_js_file";  
  
// You can also use an alias while importing a member.  
  
import Greeting as Greet from "./export.js";  
  
// If you want to import all the members but don't  
// want to Specify them all then you can do that using  
// a '*' star symbol.  
  
import * as exp from "./export.js";
```

```
import { myLogger, Alligator } from 'app.js';
```

Copy

## Importing with alias

You can also alias members at import time:

```
import myLogger as Logger from 'app.js';
```

- **Example 1:** Create a file named **import.html** and write the below code in that file.

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <title>Import in ES6</title>  
  </head>  
  <body>  
    <script type="module">  
  
      // Default member first  
      import hello, { num_set, Greeting } from "./export.js";  
      console.log(num_set);  
      hello();  
    </script>  
  </body>  
</html>
```

```

        let g = new Greeting("Aakash");
        console.log(g.greeting);
    </script>
</body>
</html>

```

- **Example 2:**

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Import in ES6</title>
  </head>
  <body>
    <script type="module">
      import * as exp from "./export.js";

      // Use dot notation to access members
      console.log(exp.num_set);
      exp.hello();
      let g = new exp.Greeting("Aakash");
      console.log(g.greeting);
    </script>
  </body>
</html>

```

**Example:**

```

//file math.js
function square(x) {
  return x * x;
}
function cube(x) {
  return x * x;
}
export { square, cube };

```

```

//while importing square function in test.js
import { square, cube } from './math;
console.log(square(8))//64
console.log(cube(8))//512

```

**Output:**

64

512

## Working with new Data Types: Enums

Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent or create a set of distinct cases.

### using `Object.freeze`

This makes our enumeration `Objects` immutable. This means that we can't add or remove properties. Nor can we update existing values for properties. This is very important if we want our enumeration to be effective

```
const ANIMALS = {  
  cat: 'cat',  
  dog: 'dog',  
  mouse: 'mouse'  
}  
Object.freeze(ANIMALS)ANIMALS.rabbit = null // Error, Object is not extensible  
ANIMALS.cat = 'dog' // Error, cannot assign to read-only property  
delete ANIMALS.mouse // Error, cannot delete property
```

```
const  
status_codes  
= {  
  200: "Success!",  
  400: "Bad Request. Make sure everything is entered correctly",  
  401: "Unauthorized; please make sure you are logged in.",  
  404: "The requested page was not found.",  
  // ...
```



## Working with for-of and for-in

Both for..in and for..of are looping constructs which are used to iterate over data structures. The only **difference between** them is the entities they iterate over: for..in iterates over all enumerable property keys of an object. for..of iterates over the values of an iterable object.

```
const array1 = ['a', 'b', 'c'];
for (const element of array1) {
  console.log(element);
}
// expected output: "a"
// expected output: "b"
// expected output: "c"
```

The **for...in** statement iterates over all [enumerable properties](#) of an object that are keyed by strings (ignoring ones keyed by [Symbols](#)), including inherited enumerable properties.

```
const object = { a: 1, b: 2, c: 3 };
for (const property in object) {
  console.log(`${property}: ${object[property]}`);
}
// expected output:
// "a: 1"
// "b: 2"
// "c: 3"
```

## TypeScript

**TypeScript** is a superset of JavaScript which primarily provides optional static typing, classes, and interfaces. One of the big benefits is to enable IDEs to provide a richer environment for spotting common errors as you type the code.

TypeScript offers all of JavaScript's features, and an additional layer on top of these:

### **TypeScript's type system.**

For example, JavaScript provides language primitives like string and number, but it doesn't check that you've consistently assigned these. TypeScript does.

This means that your existing working JavaScript code is also TypeScript code. The main benefit of TypeScript is that it can highlight unexpected behavior in your code, lowering the chance of bugs.

TypeScript checks a program for errors before execution, and does so based on the *kinds of values*, it's a *static type checker*.

### **For Example**

```
const obj = { width: 10, height: 15 };  
const area = obj.width * obj.heighth;
```

Property 'heighth' does not exist on type '{ width: number; height: number; }'. Did you mean 'height'?

```
let a = (4  
)'expected.
```

TypeScript doesn't consider any JavaScript code to be an error because of its syntax. This means you can take any working JavaScript code and put it in a TypeScript file without worrying about exactly how it is written.

Like others programming languages, TypeScript supports access modifiers at the class level. TypeScript supports three access modifiers - public, private, and protected.

1. **Public** - By default, members (properties and methods) of TypeScript class are public - so you don't need to prefix members with the public keyword. Public members are accessible everywhere without restrictions
2. **Private** - A private member cannot be accessed outside of its containing class. Private members can be accessed only within the class.
3. **Protected** - A protected member cannot be accessed outside of its containing class. Protected members can be accessed only within the class and by the instance of its sub/child class.

The `protected` modifier acts much like the `private` modifier with the exception that members declared `protected` can also be accessed within deriving classes. For example,

```
class Person {
  protected name: string;
  constructor(name: string) {
    this.name = name;
  }
}

class Employee extends Person {
  private department: string;

  constructor(name: string, department: string) {
    super(name);
    this.department = department;
  }

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in ${this.department}.`;
  }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name);
Property 'name' is protected and only accessible within class 'Person' and its subclasses.
```

Notice that while we can't use `name` from outside of `Person`, we can still use it from within an instance method of `Employee` because `Employee` derives from `Person`.

A constructor may also be marked `protected`. This means that the class cannot be instantiated outside of its containing class but can be extended. For example,

```
class Person {
  protected name: string;
  protected constructor(theName: string) {
    this.name = theName;
  }
}

// Employee can extend Person
class Employee extends Person {
  private department: string;

  constructor(name: string, department: string) {
    super(name);
  }
}
```

```

        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John");
Constructor of class 'Person' is protected and only accessible within the class declaration

```

### Readonly modifier

You can make properties readonly by using the readonly keyword. Readonly properties must be initialized at their declaration or in the constructor.

```

class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;

    constructor(theName: string) {
        this.name = theName;
    }
}

let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit";
Cannot assign to 'name' because it is a read-only property.

```

## Working with Generics

When writing programs, one of the most important aspects is to build reusable components. This ensures that the program is flexible as well as scalable in the long-term.

Generics offer a way to create reusable components. Generics provide a way to make components work with any data type and not restrict to one data type. So, components can be called or used with a variety of data types. Generics in TypeScript is almost similar to C# generics.

Generics enforce type safety without compromising performance, or productivity. In generics, a type parameter is supplied between the open (<) and close (>) brackets and which makes it strongly typed collections i.e. generics collections contain only similar types of objects. In TypeScript, you can create generic functions, generic methods, generic interfaces and generic classes.

```

function getArray(items: any[] ) : any[] {
    return new Array().concat(items);
}

let myNumArr = getArray([100, 200, 300]);
let myStrArr = getArray(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // OK
myStrArr.push(500); // OK

console.log(myNumArr); // [100, 200, 300, 400, "Hi"]
console.log(myStrArr); // ["Hello", "World", "Hello TypeScript", 500]

```

In the above example, the `getArray()` function accepts an array of type `any`. It creates a new array of type `any`, concatenates items to it and returns the new array. Since we have used type `any` for our arguments, we can pass any type of array to the function. However, this may not be the desired behavior. We may want to add the numbers to number array or the strings to the string array but not numbers to the string array or vice-versa.

To solve this, TypeScript introduced generics. Generics uses the type variable `<T>`, a special kind of variable that denotes types. The type variable remembers the type that the user provides and works with that particular type only. This is called preserving the type information.

The above function can be rewritten as a generic function as below.

Example: Generic Function

Copy

```

function getArray<T>(items: T[] ) : T[] {
    return new Array<T>().concat(items);
}

let myNumArr = getArray<number>([100, 200, 300]);
let myStrArr = getArray<string>(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // Compiler Error
myStrArr.push(500); // Compiler Error

```

In the above example, the type variable `T` is specified with the function in the angle brackets `getArray<T>`. The type variable `T` is also used to specify the type of the arguments and the return value. This means that the data type which will be specified at the time of a function call, will also be the data type of the arguments and of the return value.

We call generic function `getArray()` and pass the numbers array and the strings array. For example, calling the function as `getArray<number>([100, 200, 300])` will replace `T` with the number and so, the type of the arguments and the return value will be number array. In the same way, for `getArray<string>(["Hello", "World"])`, the type of arguments and the return value will be string array. So now, the compiler will show an error if you try to add a string in `myNumArr` or a number in `myStrArr` array. Thus, you get the type checking advantage.

It is not recommended but we can also call a generic function without specifying the type variable. The compiler will use type inference to set the value of `T` on the function based on the data type of argument values.

Example: Calling Generic Function without Specifying the Type

Copy

```
let myNumArr = getArray([100, 200, 300]); // OK
let myStrArr = getArray(["Hello", "World"]); // OK
```

**generic.ts**

```
function doReverse(list: T[]): T[] {

    let revList: T[] = [];

    for (let i = (list.length - 1); i >= 0; i--) {

        revList.push(list[i]);

    }

    return revList;

}

let letters = ['a', 'b', 'c', 'd', 'e'];

let reversedLetters = doReverse(letters); // e, d, c, b, a


let numbers = [1, 2, 3, 4, 5];

let reversedNumbers = doReverse(numbers); // 5, 4, 3, 2, 1
```

## Generic Functions

When a function implementation includes type parameters in its signature then it is called a generic function. A generic function has a type parameter enclosed in angle brackets (< >) immediately after the function name.

The following is an example of generic function:

**genericfunction.ts**

```
function doReverse(list: T[]): T[] {  
  
    let revList: T[] = [];  
  
    for (let i = (list.length - 1); i >= 0; i--) {  
  
        revList.push(list[i]);  
  
    }  
  
    return revList;  
  
}  
  
let letters = ['a', 'b', 'c', 'd', 'e'];  
  
let reversedLetters = doReverse(letters); // e, d, c, b, a  
  
  
  
  
  
  
  
let numbers = [1, 2, 3, 4, 5];  
  
let reversedNumbers = doReverse(numbers); // 5, 4, 3, 2, 1
```

Generics are purely compile-time representation; compiled JavaScript code does not have such representation. The compiled JavaScript (ES5) code for the above TypeScript code is give below:

**genericfunction.js**

```
function doReverse(list) {  
  
    var revList = [];  
  
    for (var i = (list.length - 1); i >= 0; i--) {  
  
        revList.push(list[i]);  
  
    }  
  
}
```

```

return revList;

}

var letters = ['a', 'b', 'c', 'd', 'e'];

var reversedLetters = doReverse(letters);


var numbers = [1, 2, 3, 4, 5];

var reversedNumbers = doReverse(numbers);

```

## Generic Classes

A generic class has a type parameter enclosed in angle brackets (< >) immediately after the class name. In generic class, same type parameter can be used to annotate method parameters, properties, return types, and local variables.

The following is an example of generic class:

### **genericclass.ts**

```

class ItemList

{

    private itemArray: Array;

    constructor() {

        this.itemArray = [];

    }


    Add(item: T) : void {

        this.itemArray.push(item);

    }


    GetAll(): Array {

        return this.itemArray;

```



```

    }

    }

    let fruits = new ItemList();

    fruits.Add("Apple");

    fruits.Add("Mango");

    fruits.Add("Orange");


    let listOffruits = fruits.GetAll();

    for (let i = 0; i < listOffruits.length; i++) {

        console.log(listOffruits[i]);

    }

    /* -- Output ---

    Apple

    Mango

    Orange

    */

```

Generics are purely compile-time representation; compiled JavaScript code does not have such representation. The compiled JavaScript (ES5) code for the above TypeScript code is give below:  
**genericclass.js**

```

var ItemList = (function () {

    function ItemList() {

        this.itemArray = [];

    }

    ItemList.prototype.Add = function (item) {

```

```
this.itemArray.push(item);

};

ItemList.prototype.GetAll = function () {

return this.itemArray;

};

return ItemList;

})();

var fruits = new ItemList();

fruits.Add("Apple");

fruits.Add("Mango");

fruits.Add("Orange");

var listOffruits = fruits.GetAll();

for (var i = 0; i < listOffruits.length; i++) {

    console.log(listOffruits[i]);

}
```

**Interface** is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

The TypeScript compiler does not convert interface to JavaScript. It uses interface for type checking. This is also known as "duck typing" or "structural subtyping".

An interface is defined with the keyword `interface` and it can include properties and method declarations using a function or an [arrow function](#).

Example: Interface

```
interface IEmployee {  
    empCode: number;  
    empName: string;  
    getSalary: (number) => number; // arrow function  
    getManagerName(number): string;  
}
```

In the above example, the `IEmployee` interface includes two properties `empCode` and `empName`. It also includes a method declaration `getSalary` using an arrow function which includes one number parameter and a number return type. The `getManagerName` method is declared using a normal function. This means that any object of type `IEmployee` must define the two properties and two methods.

# Angular

## Creating a new angular project

We create our angular project with the command

```
Ng new <projectname>
```

The command generates several new files and folders for us.

```
$ ng new angular-hello-world
```

## Running the angular project

```
$ ng serve
```

## Component

Components are the fundamental building block of Angular applications.

Components are *composable*, we can build larger Components from smaller ones.

An Angular application is therefore just a tree of such Components, when each Component renders, it recursively renders its children Components.

At the root of that tree is the top-level Component, the *root* Component.

When we *bootstrap* an Angular application, we are telling the browser to render that top-level *root* Component which renders its child Components and so on.

When building a new Angular application, we start by:

1. Breaking down an applications into seperate Components.
2. For each component we describe its *responsibilities*.
3. Once we've desribed the responsibilites we then describe its *inputs & outputs*, its public facing interface.

Components are a feature of Angular that let us create a new HTML *language* and they are how we structure Angular applications.

HTML comes with a bunch of pre-built tags like `<input>` and `<form>` which look and behave a certain way. In Angular we create new *custom* tags with their own look and behaviour.

An Angular application is therefore just a set of custom tags that interact with each other, we call these tags *Components*.

And if you see, this new project is bootstrapped with one component, our root component which it called `AppComponent` and has a selector of `app-root`.

We use a new feature of TypeScript called *annotations*, and specifically an annotation called `@Component`, like

The `@Component` is an annotation, an annotation automatically adds some default code to the class, function or property it's attached to.

You can configure the `@Component` annotation by passing it an object with various parameters. In our example above `@Component` has one parameter called `selector`, this tells Angular which tag to link this class too.

By setting the selector to `approot` we've told angular that whenever it finds a tag in the HTML like `<approot></approot>` to use an instance of the AppComponent class to control it.

Before we can use `@Component` though we need to import it, like so

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

The line above is saying we want to import the `Component` code from the module `@angular/core`.

## Angular Modules

In Angular your code is structured into `packages` called Angular Modules, or `NgModules` for short. Every app requires at least one module, the root module, that we call `AppModule` by convention.

They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular application has at least one NgModule class, the *root module*, which is conventionally named `AppModule` and resides in a file named `app.module.ts`. You launch your application by *bootstrapping* the root NgModule.

While a small application might have only one NgModule, most applications have many more *feature modules*. The *root* NgModule for an application is so named because it can include child NgModules in a hierarchy of any depth.

When you open this `app.module.ts` file, you can see

An NgModule is defined by a class decorated with `@NgModule()`.

The `@NgModule()` decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

- **declarations:** The components, *directives*, and *pipes* that belong to this NgModule.
- **exports:** The subset of declarations that should be visible and usable in the *component templates* of other NgModules.

- **imports:** Other modules whose exported classes are needed by component templates declared in *this* NgModule.
- **providers:** Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the application. (You can also specify providers at the component level.)
- **bootstrap:** The main application view, called the *root component*, which hosts all other application views. Only the *root NgModule* should set the bootstrap property. Identifies the root component that Angular should bootstrap when it starts the application.

We know **NgModule** but **BrowserModule** is the Angular Module that contains all the needed Angular bits and pieces to run our application in the browser. So, Almost every application's root module should import the BrowserModule.

Angular itself is split into separate Angular Modules so we only need to import the ones we really use. Some other common modules you'll see in the future are the **FormsModule**, **RouterModule** and **HttpModule**.

We also need to remember to import NgModule and BrowserModule, like

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

### Data Binding

- Data bindings are basically expressions
- They are used in templates and evaluated at run time to add dynamic content to HTML
- They are applied as attributes on DOM elements or as a sequence of characters in strings
- We can do binding in the template by using the special **{{ }}** syntax, also known as *moustache syntax*. (string interpolation)
- The **{{ }}** contains JavaScript which is run by Angular and the output inserted in the HTML.
- So if we put **{{ 1 + 1 }}** in the template the number **2** would be inserted into the HTML.
- Now that single value from class is binded to html view.
- Angular binding can be grouped into 3 categories:

Binding Type	Supported Syntax	Description
Interpolation/property /attribute	<pre>{{expression}}</pre> <pre>[(target)]="expression"</pre> <pre>bind-target="expression"</pre>	One way binding, from model to view
Event	<pre>(target)="statement"</pre> <pre>on-target="statement"</pre>	One way binding, from view to model
Two way	<pre>[(target)]="expression"</pre> <pre>bindon-target="expression"</pre>	Change to model updates view and vice versa

### Directives

- Directives power up the HTML.
- They are used to attach behavior to DOM elements
- Everything you can do with a directive you can also do with a component. But not everything you can do with a component you can do with a directive.
- Directives are just components but without views.

Angular has different types of directives: **structural** and **attribute**. As the name suggest, the attribute directives are altering the properties of an element to which they are attached and the structural ones are changing the layout, basically adding or removing elements in the *DOM*.

Angular provides a small number of built-in directives **NgFor**

- **NgIf**
- **NgSwitch**
- **NgClass**
- **NgNonBindable**

*Structural Directives* are directives which *change* the structure of the DOM by adding or removing elements.

There are three built-in structural directives, `NgIf`, `NgFor` and `NgSwitch`.

`NgFor` is a structural directive, meaning that it changes the structure of the DOM.

It's point is to repeat a given HTML template once for each value in an array, each time passing it the array value as context for string interpolation or binding.

```
@Component({
  selector: 'ngfor-example',
  template: `
<ul>
  <li *ngFor="let person of people"> (1)
    {{ person.name }}
  </li>
</ul>
`
})
class NgForExampleComponent {
  people: any[] = [
    {
      "name": "Douglas Pace"
    },
    {
      "name": "McLeod Mueller"
    },
    {
      "name": "Day Meyers"
    },
    {
      "name": "Aguirre Ellis"
    },
    {
      "name": "Cook Tyson"
    }
  ]
}
```



```
}  
];  
}
```

We loop over each `person` in the `people` array and print out the persons name.

The syntax is `*ngFor="let <value> of <collection>"`.

`<value>` is a variable name of your choosing, `<collection>` is a property on your component which holds a collection, usually an array but anything that can be iterated over in a `for-of` loop.

Sometimes we also want to get the *index* of the item in the array we are iterating over.

We can do this by adding another variable to our `ngFor` expression and making it equal to `index`, like so:

## HTML

```
<ul> (1)  
  <li *ngFor="let person of people; let i = index"> (1)  
    {{ i + 1 }} - {{ person.name }} (2)  
  </li>  
</ul>
```

We create another variable called `i` and make it equal to the special keyword `index`.

We can use the variable `i` just like we can use the variable `person` in our template.

If we ran the above we would now see this:

We use the `NgFor` directive to loop over an array of items and create multiple elements dynamically from a template element.

The *template* element is the element the directive is attached to.

We can nest multiple `NgFor` directives together.

We can get the index of the item we are looping over by assigning `index` to a variable in the `NgFor` expression.

## NgIf & NgSwitch

The **NgIf** directive is used when you want to display or remove an element based on a condition.

If the condition is **false** the element the directive is *attached to* will be *removed* from the DOM.

The difference between `[hidden]='false'` and `*ngIf='false'` is that the first method simply *hides* the element. The second method with **ngIf** *removes* the element completely from the DOM.

We define the condition by passing an expression to the directive which is evaluated in the context of its host component.

The syntax is: `*ngIf="<condition>"`

Let's use this in an example, we've taken the same code sample as we used for the **NgFor** lecture but changed it slightly. Each person now has an age as well as a name.

Let's add an **NgIf** directive to the template so we only show the element if the age is less than 30, like so:

### TypeScript

```
@Component({
  selector: 'ngif-example',
  template: `
    <h4>NgIf</h4>
    <ul *ngFor="let person of people">
      <li *ngIf="person.age < 30"> (1)
        {{ person.name }} ({{ person.age }})
      </li>
    </ul>
  `
})
class NgIfExampleComponent {

  people: any[] = [
    {
```

```

    "name": "Douglas Pace",
    "age": 35
  },
  {
    "name": "McLeod Mueller",
    "age": 32
  },
  {
    "name": "Day Meyers",
    "age": 21
  },
  {
    "name": "Aguirre Ellis",
    "age": 34
  },
  {
    "name": "Cook Tyson",
    "age": 32
  }
];
}

```

The **NgIf** directive *removes* the **li** element from the DOM if **person.age** is less than 30.

**Note: We *can't* have two structural directives, directives starting with a \*, attached to the *same* element.**

The below code would **not** work:

HTML

```

<ul *ngFor="let person of people" *ngIf="person.age < 30">
  <li>{{ person.name }}</li>
</ul>

```

## ngSwitch

Let's imagine we wanted to print people's names in different colours depending on *where* they are from. Green for UK, Blue for USA, Red for HK.

With Bootstrap we can change the text color by using the `text-danger`, `text-success`, `text-warning` and `text-primary` classes.

We *could* solve this by having a series of `*ngIf` statements, like so:

### HTML

```
<ul *ngFor="let person of people">
  <li *ngIf="person.country === 'UK'"
      class="text-success">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngIf="person.country === 'USA'"
      class="text-primary">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngIf="person.country === 'HK'"
      class="text-danger">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngIf="person.country !== 'HK' && person.country !== 'UK' && person.country !== 'USA'"
      class="text-warning">{{ person.name }} ({{ person.country }})
  </li>
</ul>
```

This initially seems to make sense until we try to create our *else* style element. We have to check to see if the person is not from any of the countries we have specified before. Resulting in a pretty long `ngIf` expression and it will only get worse the more countries we add.

Most languages, including JavaScript, have a language construct called a `switch` statement to solve this kind of problem. Angular also provides us with similar functionality via something called the `NgSwitch` directive.

This directive allows us to render different elements depending on a given condition, in fact the `NgSwitch` directive is actually a number of directives working in conjunction, like so:

```
@Component({
  selector: 'ngswitch-example',
```

```

template:`<h4>NgSwitch</h4>
<ul *ngFor="let person of people"
  [ngSwitch]="person.country"> (1)

  <li *ngSwitchCase="'UK'" (2)
    class="text-success">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'USA'"
    class="text-primary">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'HK'"
    class="text-danger">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchDefault (3)
    class="text-warning">{{ person.name }} ({{ person.country }})
  </li>
</ul>`
})

```

```

class NgSwitchExampleComponent {

```

```

  people: any[] = [
    {
      "name": "Douglas Pace",
      "age": 35,
      "country": 'MARS'
    },
    {
      "name": "McLeod Mueller",
      "age": 32,
      "country": 'USA'
    },
    {
      "name": "Day Meyers",

```

```

    "age": 21,
    "country": 'HK'
  },
  {
    "name": "Aguirre Ellis",
    "age": 34,
    "country": 'UK'
  },
  {
    "name": "Cook Tyson",
    "age": 32,
    "country": 'USA'
  }
];
}

```

1. We bind an expression to the `ngSwitch` directive.
2. The `ngSwitchCase` directive lets us define a condition which if it matches the expression in (1) will render the element it's attached to.
3. If no conditions are met in the switch statement it will check to see if there is an `ngSwitchDefault` directive, if there is it will render the element that's attached to, however it is optional — if it's not present it simply won't display anything if no matching `ngSwitchCase` directive is found.

The key difference between the `ngIf` solution is that by using `NgSwitch` we evaluate the expression only once and then choose the element to display based on the result.

## ngStyle

The `NgStyle` directive lets you set a given DOM elements style properties.

One way to set styles is by using the `NgStyle` directive and assigning it an *object literal*, like so:

### HTML

```
<div [ngStyle]="{'background-color':'green'}"></div>
```

This sets the background color of the `div` to green.

`ngStyle` becomes much more useful when the value is *dynamic*. The *values* in the object literal that we assign to `ngStyle` can be JavaScript expressions which are evaluated and the result of that expression is used as the value of the CSS property, like this:

## HTML

```
<div [ngStyle]="{'background-color':person.country === 'UK' ? 'green': 'red'}"></div>
```

The above code uses the ternary operator to set the background color to green if the persons country is the UK else red.

```
@Component({
  selector: 'ngstyle-example',
  template: `<h4>NgStyle</h4>
<ul *ngFor="let person of people">
  <li [ngStyle]="{'color':getColor(person.country)}"> {{ person.name }} ({{ person.country }}) (1)
</li>
</ul>
`
})
class NgStyleExampleComponent {

  getColor(country) { (2)
    switch (country) {
      case 'UK':
        return 'green';
      case 'USA':
        return 'blue';
      case 'HK':
        return 'red';
    }
  }

  people: any[] = [
    {

```

```

    "name": "Douglas Pace",
    "country": 'UK'
  },
  {
    "name": "McLeod Mueller",
    "country": 'USA'
  },
  {
    "name": "Day Meyers",
    "country": 'HK'
  },
  {
    "name": "Aguirre Ellis",
    "country": 'UK'
  },
  {
    "name": "Cook Tyson",
    "country": 'USA'
  }
];
}

```

We set the color of the text according to the value that's returned from the `getColor` function.

Our `getColor` function returns different colors depending on the country passed in.

As well as using the `ngStyle` directive we can also set individual style properties using the `[style.<property>]` syntax, for example `[style.color]="getColor(person.country)"`

```

<ul *ngFor="let person of people">
  <li [style.color]="getColor(person.country)">{{ person.name }} ({{ person.country }})
</li>
</ul>

```



## NgClass

---

The `NgClass` directive allows you to set the CSS class dynamically for a DOM element.

### *Tip*

The `NgClass` directive will feel very similar to what `ngClass` used to do in Angular 1.

There are two ways to use this directive, the first is by passing an object literal to the directive, like so:

```
[ngClass]='{'text-success':true}'
```

When using an object literal, the keys are the classes which are added to the element if the value of the key evaluates to true.

So in the above example, since the value is `true` this will set the class `text-success` onto the element the directive is attached to.

The value can also be an *expression*, so we can re-write the above to be.

### HTML

```
[ngClass]='{'text-success':person.country === 'UK}'
```

Let's implement the colored names demo app using `ngClass` instead of `ngStyle`.

### HTML

```
<h4>NgClass</h4>
<ul *ngFor="let person of people">
  <li [ngClass]='{'
    'text-success':person.country === 'UK',
    'text-primary':person.country === 'USA',
    'text-danger':person.country === 'HK'
  }">{{ person.name }} ({{ person.country }})
</li>
</ul>
```