

Angular 13

Trainer: Thayanithy Jegan



JAVASCRIPT A RECAP

Why Study JavaScript

JavaScript is one of the **3 languages** all web developers **must** learn:

1. **HTML** to define the content of web pages
2. **CSS** to specify the layout of web pages
3. **JavaScript** to program the behaviour of web pages

JavaScript works with both HTML and CSS. JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997. ECMAScript is official name of the language.

JavaScript Primer - Objectives

By end of chapter, everybody should know

1. Include a JavaScript on web page
2. Statements
3. Types and variables
4. Operators
5. Control structures
6. Functions, Parameters and Return Values
7. Objects, Properties and Methods
8. Arrays and JSON
9. Call-backs

Include JavaScript

There are two ways to include a JavaScript on web page

1. Referencing an External Script

It is better to keep the scripts in a separate file and create a reference to the file in the web page.

This way, the scripts can be reused.

To do this use the script tag

The src attribute points to the location of a JavaScript file.

Example

```
<script src="scripts/myScript.js"></script>
```

Include JavaScript

There are two ways to include a JavaScript on web page

2. Using an Inline Script

This method is used when the reuse of script is not necessary.

Simply for convenience

Example

```
<script>
```

```
    document.write("Hello");
```

```
</script>
```

Scripts can be placed in the `<body>`, or in the `<head>` section of an HTML page, or in both.

Output

JavaScript can "display" data in different ways:

1. Writing into an HTML element, using **innerHTML**.

```
<p id="demo"></p>
```

```
<script>
```

```
document.getElementById("demo").innerHTML = "Hello world";
```

```
</script>
```

2. Writing into the HTML output using **document.write()**.

```
<script>document.write("Hello world");</script>
```

3. Writing into an alert box, using **window.alert()**.

```
<button onclick="window.alert('Hello world')">Try it</button>
```

4. Writing into the browser console, using **console.log()**.

Node.js

Node technology has two components, such as

1. **Node.js** is an open source, cross-platform runtime environment for server-side JavaScript and a platform for building fast and scalable server applications using JavaScript. Node.js is to run JavaScript without a browser. It uses Google V8 JavaScript engine to execute code.
2. **NPM** is the Node Package Manager. NPM is a command line utility which interacts with an online repository that hosts open source Node.js projects and aids in package installation, version management and dependency management.

Node.js - Installation

Node is available here – <https://nodejs.org/en/download>

NPM node package manager, is distributed with Node.js, which means that when you install Node.js, automatically npm also get installed.

Installation on Windows

1. Download and run the .msi installer for Node.
2. To verify if the installation of node was successful, enter the command ***node --v*** in the terminal window.
3. To verify if the installation of npm was successful, enter the command ***npm --v*** in the terminal window.

Visual Studio Code - Installation

Visual Studio Code is a lightweight but powerful and FREE source code editor by Microsoft for web developers. Visual Studio Code

1. runs on desktop
2. is available for Windows, macOS and Linux

Visual Studio Code editor comes with built-in support for

1. JavaScript
2. TypeScript
3. Node.js

Node.js in Visual Studio Code

Open command window and create a folder to keep all our JavaScript code

```
mkdir JavaScript-Tutorial  
cd JavaScript-Tutorial  
code .
```

The period '.' refers to current folder. So Visual Studio Code will start and open JavaScript-Tutorial folder. Do the following steps

1. Create app.js file
2. Type JavaScript code
3. Confirm IntelliSense is working
4. Run the JavaScript in Integrated Terminal
5. Debug the JavaScript code

Statements

JavaScript program is essentially a collection of instructions to be executed.

These instructions are called statements. JavaScript statements are

1. executed line by the browser
2. executed one after the other
3. exist on its own line
4. ends with semicolon
5. multiple statements can be in one line separated with semicolon

Example

```
console.log("I am a statement");
```

```
console.log("I am also a statement");
```

Statements

1. JavaScript ignores multiple spaces in a statement.

var person = "Hege"; is same as var person="Hege";

2. For readability, avoid statements longer than 80 characters. The best place to break it, is after an operator such as “=”

document.getElementById("demo").innerHTML = "Hello Dolly!";

3. JavaScript statements can be grouped together in code blocks, inside curly brackets {...} such as if, for, do, while, function

function myFunction() {

document.getElementById("demo1").innerHTML = "Hello Dolly!";

document.getElementById("demo2").innerHTML = "How are you?";

}

Comments

Ideally, comments are used to say something useful about the purpose and context of code. The comments will help to understand why that code is there. There are two ways to comment code and they are

1. Single line comments are useful for short comments
2. Multiline comments are useful for multiline comments

Example

// The following lines are used to get user input

/ The following lines are used to validate*

User IC and Passport

*Email address */*

Values

JavaScript **syntax** is the set of rules, how JavaScript programs are constructed. The JavaScript syntax defines two types of values and they are

1. Fixed values are called **literals**.

Example: 3, 2.5, "Hello World"

2. Variable values are called **variables**.

Example: `_price`, `_discount`, `total_amount`

The JavaScript Expressions are composed of variables, operators and literals.

Variables & Identifiers

Variables are

1. the containers that hold the data.
2. named areas of computer memory in which values can be stored and retrieved.
3. declared using the keyword “var”
4. Multiple variables can be declared on a single line separated using “comma”
5. Values are assigned to a variable using the “equal” symbol

Variables & Identifiers

JavaScript variables must be identified with unique names called identifiers.

1. Identifiers can be short names or more descriptive names
2. Identifiers can contain letters, digits, underscores, and dollar signs
3. Identifiers must begin with a letter or \$ or _
4. Identifiers are case sensitive (y and Y are different variables)

```
var color = "red";  
var color, size, shape;  
var color,  
    size,  
    shape;  
color = "blue";
```

The Concept of Data Types

In programming, **data types** is an important concept.

To be able to operate on variables, it is important to know something about the variable data type. Without data types, a computer cannot safely solve the expression like

```
var x = 16 + "Volvo";
```

In JavaScript variables are dynamic data types. This means that the same variable can be used to hold different data types:

```
var x;           // Now x is undefined
```

```
var x = 5;       // Now x is a Number
```

```
var x = "John";  // Now x is a String
```

The Concept of Data Types

In JavaScript *undefined* and *null* are 2 data types that cannot contain values

In JavaScript there are 5 different data types that can contain values:

1. *number*
2. *string*
3. *boolean*
4. *object*
5. *function*

In JavaScript there are 3 types of objects:

1. *Object*
2. *Date*
3. *Array*

Undefined and Null

1. The variable is not assigned with any value

```
var myName;
```

```
console.log(myName);
```

The result: **undefined**

2. The variable is assigned with null value

```
var myName = null;
```

```
console.log(myName);
```

The result: **null**

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+	String concatenation

Assignment Operators

Operator	Description
=	Assignment
+=	Addition Assignment
-=	Subtraction Assignment
*=	Multiplication Assignment
/=	Division Assignment
%=	Modulus Assignment

Comparison Operators

Operator	Description
==	Equal to
===	Equal value and equal type
!=	Not Equal
!==	Not Equal value or not equal type
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
?	Ternary operator

Other Operators

Operator	Description
&&	Logical and
	Logical or
!	Logical not
typeof	Returns the type of a variable
Instanceof	Returns true if an object is an instance of an object type
&	Bitwise AND
	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR

Primitive Types - Numbers

JavaScript does not define different types of numbers, like integers, short, long, floating-point etc. JavaScript numbers are always stored as double precision floating point numbers.

1. `var val1 = 22;`
2. `var val2 = .5;`
3. `var val3 = 22.5;`

Can perform arithmetic operation on variables

1. `console.log(val1 + val2);`
2. `console.log(val2 + val3);`

Primitive Types - Numbers

1. **Precision** - Integers are considered accurate up to 15 digits. Floating point arithmetic is not always 100% accurate.
2. **Hexadecimal** - JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.
3. **Infinity** - Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.
4. **NaN - Not a Number** - NaN is a JavaScript reserved word indicating that a number is not a legal number.
5. **Numbers Can be Objects** - Numbers can also be defined as objects with the keyword new: **var y = new Number(123)**

Numbers Properties

Number properties belongs to the JavaScript's number object wrapper called **Number**. Number properties can only be accessed as **Number**.MAX_VALUE.

1. **MAX_VALUE** - Returns the largest number possible in JavaScript
2. **MIN_VALUE** - Returns the smallest number possible in JavaScript
3. **NaN** - Represents a "Not-a-Number" value
4. **NEGATIVE_INFINITY** - Represents negative infinity (returned on overflow)
5. **POSITIVE_INFINITY** - Represents infinity (returned on overflow)

Numbers Methods

Usually, primitive values cannot have properties and methods. But in JavaScript, primitive values are treated as objects. *Example 9.656*

1. `toString()` - returns a number as a string
2. `toExponential()`
3. `toFixed()` - a specified number of decimals [*toFixed(2) => 9.66*]
4. `toPrecision()` - a specified length [*toPrecision(2) => 9.7*]
5. `valueOf()`
6. `Number()` - returns a number
7. `parseInt()` - returns an integer
8. `parseFloat()` - returns a floating point number

Math Object

Method	Description
<code>abs(x)</code>	Returns the absolute value of x
<code>acos(x)</code>	Returns the arccosine of x, in radians
<code>asin(x)</code>	Returns the arcsine of x, in radians
<code>atan(x)</code>	Returns the arctangent of x, in radians
<code>atan2(y, x)</code>	Returns the arctangent of the quotient of its arguments
<code>ceil(x)</code>	Returns the value of x rounded up to its nearest integer
<code>cos(x)</code>	Returns the cosine of x (x is in radians)
<code>exp(x)</code>	Returns the value of E^x
<code>floor(x)</code>	Returns value of x rounded down to its nearest integer

Math Object

Method	Description
<code>log(x)</code>	Returns the natural logarithm (base E) of x
<code>max(x, y, z, ..., n)</code>	Returns the number with the highest value
<code>min(x, y, z, ..., n)</code>	Returns the number with the lowest value
<code>pow(x, y)</code>	Returns the value of x to the power of y
<code>random()</code>	Returns a random number between 0 and 1
<code>round(x)</code>	Returns the value of x rounded to its nearest integer
<code>sin(x)</code>	Returns the sine of x (x is in radians)
<code>sqrt(x)</code>	Returns the square root of x
<code>tan(x)</code>	Returns the tangent of an angle

Primitive Types - Strings

To create a string value either can use single quotation or double quotation.

1. `var firstName = "Jane";`
2. `var lastName = 'Doe';`

Using one within the other

1. `var opinion = "It's alright";`
2. `var sentence = 'Billy said, "How are you today?", and smiled.';`

Using backslash

1. `var sentence = "Billy said, \"How are you today?\", and smiled.";`
2. `var opinion = 'It\'s alright';`

Strings Methods

String methods help to work with strings.

- | | |
|------------------|-----------------|
| 1. length | ▶ replace() |
| 2. indexOf() | ▶ toUpperCase() |
| 3. lastIndexOf() | ▶ toLowerCase() |
| 4. search() | ▶ concat() |
| 5. slice() | ▶ charAt() |
| 6. substring() | ▶ charCodeAt() |
| 7. substr() | |

Primitive Types - Booleans

A Boolean variable can have either true or false. Boolean() function can be used to find out if an expression (or a variable) is true or false.

1. *var isLoggedIn = true;*
2. *var isMember = false;*

True conditions

1. *var isMember = "false";*
2. *var isMember = 1;*
3. *var isMember = "Hello";*

Primitive Types - Booleans

Comparing to other programming language the false value of a Boolean variable are quiet confusing.

False conditions

1. *var isMember = "";*
2. *var isMember = 0;*
3. *var isMember = -0;*
4. *var isMember;*
5. *var isMember = null;*
6. *var isMember = 10 / "H";*

Date

1. Displaying Date

Date();

2. Creating Date Objects

new Date()

new Date(milliseconds)

new Date(dateString)

new Date(year, month, day, hours, minutes, seconds, milliseconds)

3. Methods

toString();

toUTCString();

toDateString();

Date Methods

Method	Description
<code>getDate()</code>	Get the day as a number (1-31)
<code>getDay()</code>	Get the weekday as a number (0-6)
<code>getFullYear()</code>	Get the four digit year (yyyy)
<code>getHours()</code>	Get the hour (0-23)
<code>getMilliseconds()</code>	Get the milliseconds (0-999)
<code>getMinutes()</code>	Get the minutes (0-59)
<code>getMonth()</code>	Get the month (0-11)
<code>getSeconds()</code>	Get the seconds (0-59)
<code>getTime()</code>	Get the time (milliseconds since January 1, 1970)

Array

An array is a special variable, which can hold more than one value at a time.

Storing multiple car names in single variables could look like this:

```
var car1 = "Saab";
```

```
var car2 = "Volvo";
```

```
var car3 = "BMW";
```

This is where the Array comes in.

Using an array literal is the easiest way to create a JavaScript Array.

```
var cars = ["Saab", "Volvo", "BMW"];
```

Another way to create array is using the Array keyword.

```
var cars = new Array("Saab", "Volvo", "BMW");
```

Array

Accessing the Array Elements

```
var cars = new Array("Saab", "Volvo", "BMW");
```

An array element can be accessed by referring to the **index number**.

```
var name = cars[0];
```

```
cars[0] = "Opel";
```

Array indexes start with 0.

[0] is the first element in an array.

[1] is the second element.

The full array can be accessed by referring to the array name:

```
console.log(cars);
```

Array Properties and Methods

Array methods help to work with Arrays.

- ▶ length
- ▶ isArray()
- ▶ instanceof
- ▶ typeof
- ▶ join()
- ▶ pop()
- ▶ push()
- ▶ shift()
- ▶ unshift()
- ▶ delete
- ▶ splice()
- ▶ concat()
- ▶ toString()

Array Properties and Methods

Array methods help to work with Arrays.

1. `reverse()`
2. `sort()`

By default, the `sort()` function sorts values as strings. Because of this, `sort()` method will produce incorrect result when sorting numbers. This can be fixed by passing a compare function.

1. `Math.max.apply` to find the highest number in an array
2. `Math.min.apply` to find the lowest number in an array

Conditional Statements

1. **if** statement to specify a block of JavaScript code to be executed if a condition is true.

if (condition) {

block of code to be executed if the condition is true

}

2. **else** to specify a block of code to be executed, if the same condition is false
3. **else if** to specify a new condition to test, if the first condition is false

Conditional Statements

Switch statement to select one of many blocks of code to be executed

```
switch(expression) {  
    case n:  
        javascript code block  
        break;  
    case n:  
        javascript code block  
        break;  
    default:  
        javascript code block  
}
```

Switch expression is evaluated only once and value of the expression is compared with the values of each case.

Looping - FOR

Loops can execute a block of code a number of times.

```
for (statement 1; statement 2; statement 3) {  
    code block to be executed  
}
```

1. Statement 1 is executed before the code block starts.
2. Statement 2 defines the condition for running the code block.
3. Statement 3 is executed each time after the code block has been executed.

```
for (i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
}
```

Looping - FOR

1. **Statement 1** is used to initialize the variable used in the loop. Statement 1 is optional, it can be omitted. Many variables can be initialized in statement 1.
2. **Statement 2** is used to evaluate the condition of the initial variable. Statement 2 is also optional. If statement 2 returns true, the loop will start over again, if it returns false, the loop will end. If statement 2 is omitted, must provide a **break** inside the loop.
3. **Statement 3** increments the value of the initial variable. Statement 3 is also optional. Statement 3 can do anything like negative increment ($i--$), positive increment ($i = i + 15$), or anything else. If statement 3 is omitted code inside the loop must increment the variable.

Looping - While

The while loop loops through a block of code as long as condition is true.

```
while (condition) {  
    code block to be executed  
}
```

Example

```
while (i < 10) {  
    text += "The number is " + i;  
    i++;  
}
```

Don't forget to increase the variable used in the condition, otherwise the loop will never end. This will crash your browser.

Looping – Do / While

Do / while loop execute code block once, before checking the condition is true, then it will repeat the loop as long as the condition is true.

```
do {  
    code block to be executed  
} while (condition)
```

Example

```
do {  
    text += "The number is " + i;  
    i++;  
} while (i < 10)
```

Looping – Break / Continue

The Break Statement breaks the loop and continues executing the code after the loop

1. The break statement "jumps out" of a loop.
2. The break statement used switch() statement to "jump out".
3. The break statement can also be used to jump out of a loop.

The Continue Statement

1. The continue statement "jumps over" one iteration in loop.
2. The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

Functions

In JavaScript functions are **defined** using the keyword **function**

```
function functionName(parameters) {  
    code to be executed;  
    return value;  
}
```

1. Declared functions are not executed immediately.
2. Functions are executed only when they are invoked (called upon).
3. Semicolons are used to separate executable JavaScript statements.
4. Since a function **declaration** is not an executable statement, it is not common to end it with a semicolon.

Function - Invoking

1. Code inside a function is **not executed** when function is **defined**.
2. Code inside a function is only **executed** when the function is **invoked**.

The () operator invokes the function.

```
function functionName (param1, param2) {  
  
    javascript code;  
  
    return value;  
  
}
```

```
var result = functionName (value1, value2);
```

3. Developers to define code once and use it many times.
4. Developers can invoke the same code many times with different arguments, to produce different results.

Function Expressions

In JavaScript functions are called first class functions, because the functions are treated as an expression. Function expression can be stored in a variable.

```
var func_variable = function (param1, param2) {  
    return param1 * param2;  
};
```

```
var square_result = func_variable(3, 2);
```

1. Functions stored in variables do not need function names.
2. Functions are also called “anonymous function”, since there is no name given to the function.
3. These functions are always invoked using the variable name.

Self-Invoking Functions

1. Function expressions can be made "self-invoking".
2. A self-invoking function expression is invoked (started) automatically, without being called.

```
(function () {  
    alert ("Hello World");  
})();
```

3. The above Function expression is executed automatically since it is followed by ().
4. Parentheses is added around the function to indicate that it is a function expression

Function Parameters & Arguments

1. Function parameters are the names listed in the function definition.
2. Function arguments are real values passed to and received by function.

Parameter Rules

1. Function definitions do not specify data types for parameters.
2. Functions do not perform type checking on the passed arguments.
3. Functions do not check the number of arguments received.
4. If a function is called with missing arguments (less than declared), the missing values are set to: undefined
5. If a function is called with too many arguments (more than declared), these arguments can be reached using the arguments object.

Function Parameters & Arguments

The Arguments Object

1. JavaScript functions have a built-in object called the arguments object.
2. The argument object contains an array of the arguments used when the function was called (invoked).

Arguments are Passed by Value

1. The parameters, in a function call, are the function's arguments.
2. The function only gets to know values, not argument's locations.

Objects are Passed by Reference

1. In JavaScript, objects will behave like they are passed by reference.
2. If a function changes an object property, it changes the original value.

Objects - Introduction

In JavaScript, objects are king. In JavaScript, almost "everything" is an object.

1. Booleans can be objects (if defined with the **new** keyword)
2. Numbers can be objects (if defined with the **new** keyword)
3. Strings can be objects (if defined with the **new** keyword)
4. Dates are always objects
5. Maths are always objects
6. Regular expressions are always objects
7. Arrays are always objects
8. Functions are always objects
9. Objects are always objects

Objects - Creation

Objects are variables containing variables. Variables can contain single value.

Objects can contain many values. In object values are written as key : value pairs, where key and value are separated by colon.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

In the above example the person object contain multiple values such as firstName, lastName, age and eyeColor.

Objects are created in three different ways and they are

1. Object literal way
2. Using the keyword new
3. Function Constructor

Objects - Creation

Object Literal Way

```
var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue",  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
};
```

Object is collection of named values, called properties and methods.

Objects - Creation

Using the keyword new

```
var person = new Object();  
  
person.firstName = "John";  
  
person.lastName = "Doe";  
  
person.age = 50;  
  
person.eyeColor = "blue";  
  
person.fullName = function() {  
  
    return this.firstName + " " + this.lastName + this.age;  
  
};
```

Objects - Creation

Object Constructor

JavaScript allows to create "object type" that can be used to create many objects of one type. To create "object type" use object constructor function.

```
function Person(first, last, age, eye) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eye;  
}
```

```
var myFather = new person("John", "Doe", 50, "blue");
```

Constructor function creates the **prototype** for Person object.

Objects - Property

1. Syntax for accessing the property of an object is:

objectName.property // person.age

objectName["property"] // person["age"]

objectName[expression] // x = "age"; person[x]

2. New properties can be added to an existing object by giving it a value.

object.property = "English";

3. The delete keyword deletes a property from an object:

delete object.property;

Objects - Method

Methods are actions that can be performed on objects. A JavaScript method is a property containing a function definition.

Syntax of an object method

```
methodName : function() {  
  
    code lines  
  
}
```

An object method can be accessed with the following syntax:

```
objectName.methodName()
```

Objects - Adding

1. Add new properties/methods to an existing object.

```
myFather.nationality = "English";  
  
myFather.name = function () {  
    return this.firstName + " " + this.lastName;  
};
```

2. Add new properties/methods using prototype keyword.

```
Person.prototype.nationality = "English";  
  
Person.prototype.name = function() {  
    return this.firstName + " " + this.lastName;  
};
```

Objects – Indexed Properties

JavaScript supports objects with indexed properties. An Indexed property

1. is a property with an index number instead of a property name.
2. can not be accessed with the dot (.) operator.
3. can be accessed with the associate array operator (['index']).
4. can also be accessed with the array operator ([index]).
5. can be assigned with a function to become an indexed method.

Objects – Indexed Properties

```
var myBook = {  
    author: "Herong Yang",  
    title: "JavaScript Tutorials",  
    3: 2008,  
    price: "Free",  
    4: myToString  
};  
myBook[9] = "Programming";  
document.writeln("myBook['author']: " + myBook['author']);  
document.writeln("myBook[3]: " + myBook[3]);  
document.writeln("myBook['3']: " + myBook['3']);  
document.writeln("myBook[4]: " + myBook[4]);  
document.writeln("myBook[9]: " + myBook[9]);
```

Looping – FOR / IN

1. JavaScript for/in loops through properties of an object.
2. Block of code inside loop will be executed once for each property.

```
var person = {fname:"John", lname:"Doe", age:25};
```

```
var text = "";
```

```
var x;
```

```
for (x in person) {
```

```
    text += person[x];
```

```
}
```

```
console.log(text);
```

3. Do not use for/in statement to loop through arrays where index order is important. Use the for statement instead.

Call or Apply Method

The `call()` or `apply()` methods are used to invoke a function with an owner object as first argument. This code calls `fullName` function of `person` object by sending the `myObject` as parameter.

```
var person = {  
    firstName: "John",  
    fullName: function() {  
        return this.firstName;  
    }  
}  
  
var myObject = {  
    firstName: "Mary",  
}  
  
person.fullName.call(myObject); // Will return "Mary"
```

typeof Operator

1. `typeof "John"` // Returns "string"
2. `typeof 3.14` // Returns "number"
3. `typeof NaN` // Returns "number"
4. `typeof false` // Returns "boolean"
5. `typeof [1,2,3,4]` // Returns "object"
6. `typeof {name:'John', age:34}` // Returns "object"
7. `typeof new Date()` // Returns "object"
8. `typeof function () {}` // Returns "function"
9. `typeof myCar` // Returns "undefined" *
10. `typeof null` // Returns "object"

Constructor Property

1. `"John".constructor` // Returns function String()
2. `(3.14).constructor` // Returns function Number()
3. `false.constructor` // Returns function Boolean()
4. `[1,2,3,4].constructor` // Returns function Array()
5. `{name:'John',age:34}.constructor`
// Returns function Object()
6. `new Date().constructor` // Returns function Date()
7. `function () {}.constructor` // Returns function

Scope

Local Variables

1. Variables declared within a JavaScript function, become **LOCAL** to the function.
2. Local variables have **local scope** meaning they can only be accessed within the function.

Global Variables

1. A variable declared outside a function, becomes **GLOBAL**.
2. A global variable has **global scope** meaning all scripts and functions on a web page / java script file can access it.

Closures


Closure has 3 scope chains and they are

1. it can access to its own scope
2. it can access to outer function's variables
3. it can access to global variables

```
function showName (firstName, lastName) {  
    var nameIntro = "Your name is ";  
    function makeFullName () {  
        // access local & outer variable including parameters  
        return nameIntro + firstName + " " + lastName;  
    }  
    return makeFullName ();  
}  
showName ("Michael", "Jackson");
```



TYPESCRIPT



JavaScript that scales

Introduction

JavaScript is a language for the client side. The development of Node.js has marked JavaScript as an emerging server-side technology. The known problems with JavaScript language are

1. When the code grows, it tends to get messier
2. When the code grows, it is difficult to maintain and reuse
3. The language fails to embrace the features of Object Orientation
4. The language doesn't provide strong type checking
5. The language doesn't support compile-time error checks

TypeScript was presented to bridge this gap.

Introduction

By definition, TypeScript is nothing but JavaScript for application-scale development. TypeScript is

1. strongly typed
2. object oriented
3. decorators
4. compiled language

TypeScript was designed by **Anders Hejlsberg** (designer of C#) at Microsoft. TypeScript is both language and set of tools. TypeScript is a typed superset of JavaScript compiled to JavaScript. In short, TypeScript is JavaScript plus some additional features.

Features

1. TypeScript starts with JavaScript - All JavaScript code is TypeScript code. JavaScript code can be simply copy and paste into a TypeScript file. All JavaScript libraries work perfectly with TypeScript.
2. TypeScript ends with JavaScript - TypeScript code will be compiled to JavaScript, which means compiled TypeScript can be consumed from any JavaScript console. Runs in any browser or host on any OS.
3. JavaScript is TypeScript - This means that any valid **.js** file can be renamed to **.ts** and compiled with other TypeScript files.
4. TypeScript is portable - TypeScript is portable across browsers, devices, and operating systems. Basically TypeScript can run on any environment that JavaScript runs on.

Benefits

1. Compilation – JavaScript is an interpreted language. Since there is no compilation process, difficult to find bugs in the code. TypeScript *transpiler* provides error-checking feature.
2. Strong Static Typing – JavaScript is not strongly typed. TypeScript comes with an optional static typing and type inference system through TLS (TypeScript Language Service).
3. TypeScript **supports type definitions** for existing JavaScript libraries. TypeScript Definition file (with **.d.ts** extension) provides definition for external JavaScript libraries. Hence, TypeScript code can contain these libraries.
4. TypeScript **supports Object Oriented Programming**.

Components

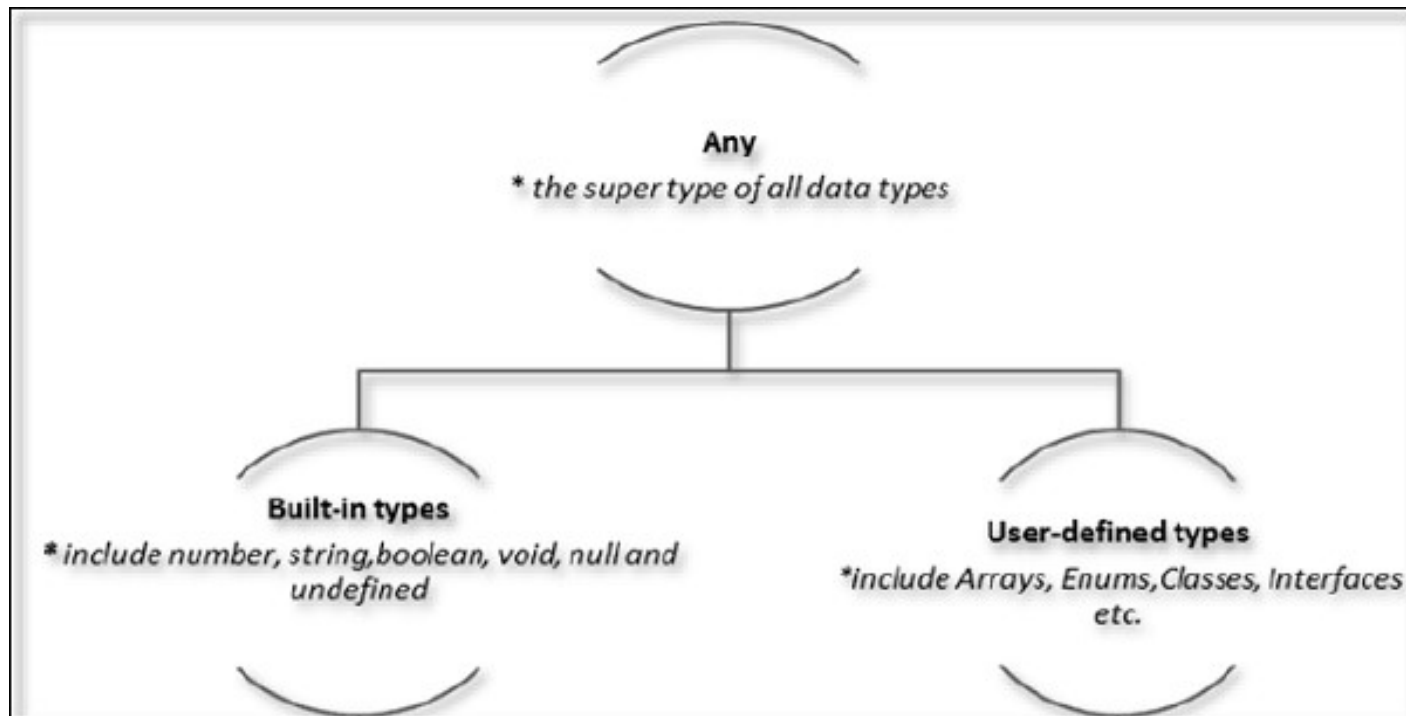
1. **Language** – Comprises of syntax, keywords and type annotations.
2. **TypeScript Compiler (tsc)** – Converts the instructions written in TypeScript to its JavaScript equivalent.
3. The TypeScript Language Service – The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications.
4. The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.

TypeScript - Installation

1. To install the latest TypeScript version open the terminal window and type **npm install -g typescript**
2. To verify if the installation of TypeScript was successful, enter the command **tsc --v** in the terminal window.
3. TypeScript is installed in the following directory
C:\Program Files (x86)\Microsoft SDKs\TypeScript
4. If the PC has multiple instances of TypeScript then to find which compiler is being used, in the terminal window type
where tsc

Type

Type System represents different types of values supported by the language. Type System checks validity of the supplied values, before they are stored or manipulated.



Basic Types

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

```
let isDone: boolean = false;
```

Number

All numbers in TypeScript are floating point values.

```
let decimal: number = 6;
```

```
let amount: number = 2.5;
```

```
let hex: number = 0xf00d;
```

```
let binary: number = 0b1010;
```

```
let octal: number = 0o744;
```

Basic Types

String

TypeScript uses double quotes (") or single quotes (') to surround string data.

```
let color: string = "blue";
```

Template strings

TypeScript supports multiple lines and embedded expressions. These strings are surrounded by backtick (`) and embedded expressions are of the form `\${ expr }`.

```
let fullName: string = `Bob Bobbington`;
```

```
let age: number = 37;
```

```
let sentence: string = `I am ${fullName}, ${age + 1} years old.`;
```

Basic Types

Array

TypeScript allows to work with arrays of values. Array types can be written in one of two ways.

```
let list: number[] = [1, 2, 3];
```

```
let list: Array<number> = [1, 2, 3];
```

Tuple

Tuple allow to express an array where the type of a fixed number of elements is known, but need not be the same.

```
let x: [string, number];
```

```
x = ["hello", 10];    =>    OK
```

```
x = [10, "hello"];    =>    Error
```


Basic Types

Tuple

When accessing an element **with an index**, correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
```

```
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

When accessing an element **outside the set of known indices**, a union type is used instead:

```
x[3] = "world"; // 'string' can be assigned to 'string | number'
```

```
console.log(x[5].toString()); // 'string' and 'number' both have  
'toString'
```

```
x[6] = true; // Error, 'boolean' isn't 'string | number'
```

Basic Types

Enum

An enum is a way of giving names to sets of numeric values.

```
enum Color {Red, Green, Blue}
```

```
let c: Color = Color.Green;
```

By default, enums begin numbering their members starting at 0. This can be changed by manually setting value to one of its members.

```
enum Color {Red = 1, Green, Blue}
```

```
let c: Color = Color.Green;
```

```
enum Color {Red = 1, Green = 2, Blue = 4}
```

```
let c: Color = Color.Green;
```

Basic Types

Enum

Enums allows to go to name from a numeric value. For example, if 2 is assigned to a color and to know the color that is mapped to value 2

```
enum Color {Red = 1, Green, Blue}  
let colorName: string = Color[2];  
alert(colorName); // Displays 'Green' as its value is 2 above
```

Any

To describe the type of variables which are not known.

```
let notSure: any = 4;  
notSure = "maybe a string instead";  
notSure = false; // okay, definitely a boolean
```

Basic Types

Any

Any type is also handy in handling array, where the array has a mix of different data types:

```
let list: any[] = [1, true, "free"];  
list[1] = 100;
```

Any type is a powerful way to work with existing JavaScript, allows to opt-in and opt-out of type-checking. But Any is not same as Object.

```
let notSure: any = 4;  
notSure.ifExists(); // okay, ifExists might exist at runtime  
notSure.toFixed(); // okay, toFixed exists (the compiler doesn't check)  
let prettySure: Object = 4;  
prettySure.toFixed(); // Error: 'toFixed' doesn't exist on 'Object'.
```

Basic Types

Void

The void is opposite of any: the absence of having any type at all. This is used as return type of functions that do not return a value:

```
function warnUser(): void {  
    alert("This is my warning message");  
}
```

Declaring variables of type void is not useful because to those variables the values undefined or null only can be assigned

```
let unusable: void = undefined;
```

Basic Types

Null and Undefined

In TypeScript, both undefined and null actually have their own types named undefined and null respectively.

```
let u: undefined = undefined;
```

```
let n: null = null;
```

By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

Basic Types

Never

The never type represents the type of values that never occur. For instance, a function that always throws an exception.

```
function error(message: string): never {  
    throw new Error(message);  
}
```

```
function infiniteLoop(): never {  
    while (true) { }  
}
```

Basic Types

Type assertions

A type assertion is like a type cast where the type of some entity could be more specific than its current type. Type assertions have two forms.

One is the “angle-bracket” syntax:

```
let someValue: any = "this is a string";
```

```
let strLength: number = (<string>someValue).length;
```

And the other is the as-syntax:

```
let someValue: any = "this is a string";
```

```
let strLength: number = (someValue as string).length;
```


Variable

A variable, by definition, is “a named space in the memory” that stores values. In other words, it acts as a container for values in a program.

TypeScript variables must follow the naming rules

1. Variable names can contain alphabets and numeric digits.
2. They cannot contain spaces and special characters, except the underscore (_) and the dollar (\$) sign.
3. Variable names cannot begin with a digit.

A variable must be declared before it is used.

The keyword var

Syntax for declaring a variable in TypeScript is to use a colon (:) after variable name, followed by its type. The keyword var is used.

1. Declare its type and value in one statement.

```
var name:string = "mary";
```

2. Declare its type but no value. Variable will be set to undefined.

```
var name:string;           [value is set to undefined]
```

3. Declare its value but no type. The variable type will be set to any.

```
var name = "mary";        [variable type is set to any]
```

4. Declare neither value nor type.

```
var name;
```

The keyword var

Scoping rules

```
if (shouldInitialize) {  
  
    var x = 10;  
  
}
```

1. The variable x was declared within the if block, and yet it is able to access from outside that block.
2. The var declarations are accessible anywhere within their containing function, module, namespace, or global scope regardless of the containing block.
3. These scoping rules can cause several types of mistake.

The keyword var

Variable capturing quirks

```
for (var i = 0; i < 10; i++) {  
    setTimeout(function() { console.log(i); }, 100 * i);  
}
```

The output will be 10, 10, 10, 10, 10, 10, 10, 10, 10, 10

The expected output 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

The setTimeout will run a function after some number of milliseconds, but only after the for loop has stopped executing; By the time the for loop has stopped executing, the value of i is 10. So each time the given function gets called, it will print out 10!

The keyword let

To overcome the problem `let` was introduced. Apart from the keyword used, `let` statements are written the same way `var` statements are.

```
let hello:number = "Hello!";
```

Block-scoping

When a variable is declared using `let`, it uses what some call *lexical-scoping* or *block-scoping*.

```
if (true) {  
    let b:number = a + 1;  
}  
console.log(b)
```

The keyword let

Re-declarations

With var declarations, if a variable declared many times it always refer to the same one.

```
function f(x) {  
    var x;  
    var x;  
}
```

Using the keyword let declarations are not as forgiving.

```
let x = 10;  
let x = 20; // error: can't re-declare 'x' in the same scope
```

The keyword let

Shadowing

```
for (let i = 0; i < matrix.length; i++) {  
  let currentRow = matrix[i];  
  for (let i = 0; i < currentRow.length; i++) {  
    sum += currentRow[i];  
  }  
}
```

This version of the loop will actually perform the summation correctly because the inner loop's *i* shadows *i* from the outer loop.

Shadowing should usually be avoided in the interest of writing clearer and bug free code.

The keyword let

Block-scoped variable capturing

The keyword let declarations have drastically different behaviour when declared as part of a loop.

```
for (let i = 0; i < 10 ; i++) {  
    setTimeout(function() { console.log(i); }, 100 * i);  
}
```

In TypeScript using the let keyword in the for loop captures the state of a variable without creating the IIFE for every iteration of the for loop.

The keyword const

The const declarations are another way of declaring variables.

```
const numLivesForCat = 9;
```

They are like let declarations but, as their name implies, their value cannot be changed once they are bound.

```
const numLivesForCat = 9;  
const kitty = {  
    name: "Aurora",  
    numLives: numLivesForCat,  
}
```

```
// Error  
kitty = {  
    name: "Danielle", numLives:  
    numLivesForCat  
};
```

```
// all "okay"  
kitty.name = "Rory";  
kitty.name = "Kitty";
```

Array destructuring

The simplest form of destructuring is array destructuring assignment:

```
let input = [1, 2];  
let [first, second] = input;  
console.log(first); // outputs 1  
console.log(second); // outputs 2
```

Create a variable for the remaining items in a list using the syntax ...:

```
let [first, ...rest] = [1, 2, 3, 4];  
console.log(first); // outputs 1  
console.log(rest); // outputs [ 2, 3, 4 ]
```

Ignore trailing elements

```
let [first] = [1, 2, 3, 4];  
console.log(first); // outputs 1
```

Compile and Execute

1. **Step 1** – Create a file and save the file with .ts extension. Let us save the file as helloworld.ts. The code editor marks errors in the code, if any, while the file is being saved.
2. **Step 2** – Right-click the TypeScript file under the working Files option in VS Code's Explore Pane. Select the "Open in Command Prompt" option.
3. **Step 3** – To compile the file use the following command on the terminal window ***tsc helloworld.ts***
4. **Step 4** – The file is compiled to Test.js. To run the program written, type the following in the terminal window ***node helloworld.js***
5. **Note** - Multiple files can be compiled using the following style
tsc file1.ts, file2.ts, file3.ts

Type Assertion

TypeScript allows changing a variable from one type to another. TypeScript refers to this process as *Type Assertion*.

The syntax is to put the target type between < > symbols and place it in front of the variable or expression.

```
var str = '1'  
var str2:number = <number> str  
console.log(str2)
```

It is not called "type casting". Type casting generally implies runtime support while, "type assertions" are compile time construct and a way to provide hints to compiler on the code to be analysed.

Inferred Typing

1. Typescript is strongly typed, but this feature is optional.
2. TypeScript also allow dynamic typing of variables.
3. TypeScript encourages declaring a variable without a type.
4. The compiler will determine the type of the variable on the basis of the value assigned to it.

```
var num = 2; // data type inferred as number
```

```
console.log("value of num "+num);
```

```
num = "12";
```

```
console.log(num)
```

Variable Scope

Scope of a variable specifies where variable is defined. The availability of a variable within a program is determined by its scope.

1. **Global Scope** – Global variables are declared outside the programming constructs. Variables can be accessed from anywhere within your code.
2. **Class Scope** – Variables are also called **fields**. Fields or class variables are declared within the class but outside the methods. These variables can be accessed using the object of the class. Fields can also be static. Static fields can be accessed using the class name.
3. **Local Scope** – Local variables, as the name suggests, are declared within the constructs like methods, loops etc. Local variables are accessible only within the construct where they are declared.

Variable Scope

The following example illustrates variable scopes in TypeScript.

```
var global_num = 12;                                //global variable
class Numbers {
        num_val = 13;                                    //class variable
        static sval = 10;                                //static field
        storeNum():void {
                var local_num = 14;                    //local variable
        }
}
console.log("Global num: "+global_num);
console.log(Numbers.sval);                            //static variable
var obj = new Numbers();
console.log("Global num: "+obj.num_val);
```

Operator

An operator defines some function that will be performed on the data.

Major operators in TypeScript can be classified as

1. Arithmetic operators
2. Logical operators
3. Relational operators
4. Bitwise operators
5. Assignment operators
6. Ternary/conditional operator
7. String operator
8. Type Operator

Arithmetic Operators

Operator	Description	Example
+ (Addition)	returns the sum of the operands	a + b is 15
- (Subtraction)	returns the difference of the values	a - b is 5
* (Multiplication)	returns the product of the values	a * b is 50
/ (Division)	performs a division operation and returns quotient	a / b is 2
% (Modulus)	performs a division and returns the remainder	a % b is 0
++ (Increment)	Increments the value of the variable by one	a++ is 11
-- (Decrement)	Decrements the value of the variable by one	a-- is 9

Logical Operators

Operator	Description	Example
&& (And)	The operator returns true only if all the expressions specified return true	(A > 10 && B > 10) is False
(OR)	The operator returns true if at least one of the expressions specified return true	(A > 10 B > 10) is True
! (NOT)	The operator returns the inverse of the expression's result. For E.g.: !(>5) returns false	!(A > 10) is True

Relational Operators

Operator	Description	Example
>	Greater than	(A > B) is False
<	Lesser than	(A < B) is True
>=	Greater than or equal to	(A >= B) is False
<=	Lesser than or equal to	(A <= B) is True
==	Equality	(A == B) is false
!=	Not equal	(A != B) is True
>	Greater than	(A > B) is False

Bitwise Operators

Assume variable A = 2 and B = 3

Operator	Example
& (Bitwise AND)	(A & B) is 2
(BitWise OR)	(A B) is 3
^ (Bitwise XOR)	(A ^ B) is 1
~ (Bitwise Not)	(~B) is -4
<< (Left Shift)	(A << 1) is 4
>> (Right Shift)	(A >> 1) is 1
>>> (Right shift with Zero)	(A >>> 1) is 1

Assignment Operators

Operator	Example
= (Simple Assignment)	$C = A + B$ will assign the value of $A + B$ into C
$+=$ (Add and Assignment)	$C += A$ is equivalent to $C = C + A$
$-=$ (Subtract and Assignment)	$C -= A$ is equivalent to $C = C - A$
$*=$ (Multiply and Assignment)	$C *= A$ is equivalent to $C = C * A$
$/=$ (Divide and Assignment)	$C /= A$ is equivalent to $C = C / A$

Miscellaneous Operators

1. The negation operator (-) - Changes the sign of a value

```
var x:number = 4
```

```
var y = -x;
```

2. String Operators: Concatenation operator (+)

```
var msg:string = "hello"+"world"
```

```
console.log(msg)
```

3. Conditional Operator (?)

```
Test ? expr1 : expr2
```

4. typeof operator

```
var num = 12 console.log(typeof num);
```

5. instanceof

Decision Making

Decision-making structures requires one or more conditions to be evaluated along with a statement or statements to be executed if condition is determined to be true, and optionally, other statements to be executed if condition is false.

S. No.	Statement & Description
1.	if statement An 'if' statement consists of a Boolean expression followed by one or more statements.
2.	if...else statement An 'if' statement can be followed by an optional 'else' statement, which executes when the Boolean expression is false.
3.	else...if and nested if statements You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
4.	switch statement A 'switch' statement allows a variable to be tested for equality against a list of values.

Decision Making

if..then

```
if (conditional) {  
    //do something  
} else {  
    //do another job  
}
```

switch..case

```
switch (option) {  
    case option1:  
        // do option1 job  
        break;  
    case option2:  
        // do option2 job  
        break;  
}
```


Loops

Loop statement execute a group of statements multiple times.

Definite Loop

S. No.	Statement & Description
1.	for loop The for loop is an implementation of a definite loop.

Indefinite Loop

S. No.	Statement & Description
1.	while loop The while loop executes the instructions each time the condition specified evaluates to true.
2.	do... while The do...while loop is similar to while loop except that the do...while loop doesn't evaluate condition for the first time

Loops

Iteration operation is useful when we do repetitive activities.

For

```
for (var counter=0; counter<0; counter++) {  
    console.log(counter);  
}
```

While

```
var num = 0;  
while (num<10) {  
    console.log(num);  
    num++;  
}
```

Break & Continue

Break statement is used to take control out of a construct. Using break in a loop causes program to exit the loop.

```
var i:number = 1;
while (i<=10) {
    if (i % 5 == 0) {
        console.log (i) ;
        break;
    }
    i++;
}
```

Continue statement skips the subsequent statements in current iteration and takes control back to the beginning of the loop.

Function Definition & Invocation

Functions are the fundamental building block of any application in JavaScript. A function definition specifies what and how a task to be done.

Functions are defined using the function keyword.

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

Function Invocation - A function must be called so as to execute it. This process is termed as function invocation.

```
function test() {  
    // function definition  
    console.log("function called")  
}  
test();
```

Returning Functions

Functions may also return value along with control, back to the caller. They are called returning functions.

```
function function_name():return_type {  
    // function body  
    return value;  
}
```

1. The return_type can be any valid data type.
2. A returning function must end with a return statement.
3. A function can return at the most one value only.
4. Data type of value returned must match return type of function.

Parameterized Function

Parameters are a mechanism to pass values to functions.

S. No.	Statement & Description
1.	Call by value: This method copies actual value of an argument into formal parameter of the function. In this case, changes made to parameter inside the function have no effect on the argument.
2.	Call by pointer: This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter

Typing the Parameters

TypeScript allows to add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so return type is optional.

```
function add(x: number, y: number): number {  
    return x + y;  
}
```

Functions can refer to variables outside of the function body. When they do so, they're said to capture these variables.

```
let z = 100;  
function addToZ(x, y) {  
    return x + y + z;  
}
```

Positional Parameters

Typing the Parameters forces the data type of the value being passed must match the type of the parameter during its declaration.

```
function show (ic_number:number, student_name:string, description) {  
    console.log(student_name + " [ " + ic_number + " ] ");  
    console.log(description);  
}  
test_param(761502, "Thayanithy Jegan", "myphoto.jpg");
```

1. Code declares a function show with three parameters.
2. In absence of a data type, the parameter is considered to be of type *any*.

In the above example, the third parameter will be of the type *any*.

3. Data type of value passed must match of parameter type.

Optional Parameters

Optional parameters can be used when arguments need not be compulsorily passed for a function's execution. A parameter can be marked optional by appending a question mark to its name.

```
function student_details(id:number, name:string, mail_id?:string) {  
    console.log("ID:", id);  
    console.log("Name", name);  
    if(mail_id!=undefined) console.log("Email Id", mail_id);  
}  
  
student_details(123, "John");  
student_details(111, "mary", "mary@xyz.com");
```

Any optional parameters must follow the required parameters. If first name optional rather than last name, change the order of parameters.

Default Parameters

Function parameters can also be assigned values by default. However, such parameters can also be explicitly passed values.

```
function calculate_discount (price:number, rate:number = 0.50) {  
  let discount = price * rate;  
  console.log("Discount Amount: ", discount);  
}  
  
calculate_discount (1000)  
calculate_discount (1000, 0.30)
```

Same function is invoked with two arguments. The default value of rate is overwritten and is set to the value explicitly passed when the function is called with second parameter.

Rest Parameters

Rest parameters don't restrict number of values that can be passed to a function. All the values passed must be of same type. To declare a rest parameter, parameter name is prefixed with three periods.

```
function addNumbers(...nums:number[]) {  
    var i;  
    var sum:number = 0;  
    for(i = 0;i<nums.length;i++) {  
        sum = sum + nums[i];  
    }  
    console.log("sum of the numbers",sum)  
}  
  
addNumbers(1,2,3)  
addNumbers(10,10,10,10,10)
```

Anonymous Function

Functions declared without a name is called as anonymous functions. This function is assigned to a variable and is called a function expression.

```
let multi = function (a:number, b:number) {  
    return a*b;  
};  
console.log(multi(12, 2));
```

Anonymous function returns the product of the values passed to it. One common use for anonymous functions is as arguments to other functions.

```
function arithmetic (a:number, b:number, customFunction) {  
    return customFunction(a, b);  
};  
console.log(arithmetic(12, 2, multi));
```

Function Constructor

TypeScript supports built-in JavaScript constructor called Function (). The Function constructor creates a new Function object dynamically,

Syntax

```
var res = new Function(Arg1, Arg2..., "Function Body");
```

Function() constructor expects any number of string arguments. The last argument is the body of the function – it can contain arbitrary JavaScript statements, separated from each other by semicolons.

```
var myFunction = new Function("a", "b", "return a * b");  
var x = myFunction(4, 3);  
console.log(x);
```

The new Function() is a call to the constructor which in turn creates and returns a function reference.

Function Constructor

The Functions created using Function constructor do not create closures to their creation contexts, they always created in the global scope.

```
var x = 10;  
function createFunction1() {  
    var x = 20;  
    return new Function('return x;'); // this |x| refers global |x|  
}  
function createFunction2() {  
    var x = 20;  
    function f() {  
        return x; // this |x| refers local |x| above  
    }  
    return f;  
}
```

Lambda Functions

Lambda refers to anonymous functions in programming. These functions are also called as arrow functions. There are 3 parts to a Lambda function

1. Parameters – A function may optionally have parameters
2. The fat arrow notation/lambda notation (\Rightarrow) – It is also called as the goes to operator
3. Statements – Represent the function's instruction set

Tip – By convention, the use of single letter parameter is encouraged for a compact and precise function declaration.

Lambda Expression

Lambda Expression is an anonymous function expression that points to a single line of code. Its syntax is as follows

([param1, param2,...param n]) => statement;

Example: Lambda Expression

let foo = (x:number) => 10 + x
console.log(foo(100)) //outputs 110

The code declares a lambda expression function. The function returns sum of 10 and argument passed. On compiling, TypeScript generates JavaScript

var foo = function (x) {
 return 10 + x;
};
console.log(foo(100));

Lambda Statement

Lambda statement is an anonymous function declaration that points to a block of code. This is used when function body spans multiple lines.

```
( [param1, param2,...param n] )=> {  
    //code block  
}
```

Example: Lambda statement

```
let foo = (x:number)=> {  
    x = 10 + x;  
    console.log(x);  
}  
foo(100);
```

The function's reference is returned and stored in the variable foo.

Lambda - Syntactic Variations

It is not mandatory to specify the data type of a parameter.

```
let foo = (x)=> {  
  console.log(x);  
}
```

Optional parentheses for a single parameter

```
let foo = x => {  
  console.log(x);  
}
```

Empty parentheses for no parameter

```
let foo = () => {  
  console.log("Hello");  
}
```

Lambda and 'this'

In JavaScript, `this` is a variable that's set when a function is called. This makes it a very powerful and flexible feature. This is notoriously confusing, especially when returning a function or passing a function as an argument.

```
let students = {  
    names: ["Jegan", "John", "Azam", "David"],  
    createStudentName: function() {  
        return function() {  
            return this.names[0];  
        }  
    }  
}  
  
let studentPicker = students.createStudentName();  
let student = studentPicker();
```

It says `this.names` is not valid

Problem occurs because the context is created during the execution time not at the coding time.

Lambda and 'this'

To fix this make sure the function is bound to correct context. Arrow functions capture "this" where the function is created rather than where it is invoked.

```
let students = {  
  names: ["Jegan", "John", "Azam", "David"],  
  createStudentName: function() {  
    return () => {  
      return this.names[0];  
    }  
  }  
}  
  
let studentPicker = students.createStudentName();  
let student = studentPicker();  
console.log("Student: " + student);
```

Function Overloads

Functions with same name have capability to operate differently on basis of input provided to them. This is called Function Overloading.

Different data type of the parameter

```
function disp(string):void;
```

```
function disp(number):void;
```

The number of parameters

```
function disp(n1:number):void;
```

```
function disp(x:number,y:number):void;
```

The sequence of parameters

```
function disp(n1:number,s1:string):void;
```

```
function disp(s:string,n:number):void;
```

Classes

TypeScript is object oriented JavaScript which supports object oriented programming features like classes, interfaces, etc. Use the class keyword to declare a class in TypeScript.

```
class class_name { }
```

1. **Fields** – A field is any variable declared in a class. Fields represent data pertaining to objects.
2. **Constructors** – Responsible for allocating memory for objects of the class.
3. **Functions** – Functions represent actions an object can take. They are also at times referred to as methods.

These components are termed as the data members of the class.

Declaring a Class

```
class Car {  
    engine:string;  
    constructor(engine:string) { this.engine = engine }  
    disp():void { console.log("Engine is : "+this.engine) }  
}
```

-----Generated JavaScript Code -----

```
var Car = (function () {  
    function Car(engine) { this.engine = engine; }  
    Car.prototype.disp = function () {  
        console.log("Engine is : " + this.engine);  
    };  
    return Car;  
})();
```

Declaring a Class

1. Car class has a field named engine. The **var** keyword is not used while declaring a field.
2. A constructor is a special function of the class that is responsible for initializing the variables of the class. TypeScript defines a constructor using the constructor keyword.
3. The **this** keyword refers to the current instance of the class.
4. *disp()* is a simple function definition. Note that the **function** keyword is also not used here.

Creating Instance Objects

//create an object

var obj = new Car("XXSY1")

//access the field and function

console.log("Reading attribute value Engine as : "+obj.engine)

obj.disp()

-----Generated JavaScript Code -----

//create an object

var obj = new Car("XXSY1");

//access the field and function

console.log("Reading attribute value Engine as : " + obj.engine);

obj.disp();

Data Hiding

Object Orientation uses concept of access modifiers or access specifiers to implement concept of Encapsulation. The access specifiers/modifiers define visibility of a class's data members outside its defining class.

S. No.	Access Specifier & Description
1.	public: A public data member has universal accessibility. Data members in a class are public by default.
2.	private: Private data members are accessible only within the class that defines these members. If an external class member tries to access a private member, the compiler throws an error.
3.	protected: A protected data member is accessible by the members within the same class as that of the former and also by the members of the child classes.

Data Hiding

Let us now take an example to see how data hiding works

```
class Encapsulate {  
    str:string = "hello"  
    private str2:string = "world"  
}  
  
var obj = new Encapsulate()  
console.log(obj.str)  
console.log(obj.str2)
```

The class has two string attributes, str1 and str2, which are public and private members respectively. The class is instantiated.

The example returns a compile time error, as the private attribute str2 is accessed outside the class that declares it.

Single Inheritance

Inheritance is ability to create new classes from an existing class.

```
class Shape {
```

```
    Area:number
```

```
    constructor(a:number) { this.Area = a }
```

```
}
```

```
class Circle extends Shape {
```

```
    disp():void { console.log("Area of the circle: "+this.Area) }
```

```
}
```

```
var obj = new Circle(223);
```

```
obj.disp()
```

Multi-level Inheritance

The parent class is Root which is extended by Child and the child is again extended by Leaf Class. This is termed as **multi-level inheritance**

```
class Root {  
    str:string;  
}  
class Child extends Root {  
}  
class Leaf extends Child {  
}  
var obj = new Leaf();  
obj.str = "hello"  
console.log(obj.str)
```

Method Overriding

Method Overriding allows child class redefines the superclass's method.

```
class PrinterClass {  
    doPrint():void { console.log("doPrint() from Parent called...") }  
}  
  
class StringPrinter extends PrinterClass {  
    doPrint():void {  
        super.doPrint()  
        console.log("doPrint() is printing a string...")  
    }  
}  
  
var obj = new StringPrinter()  
obj.doPrint()
```

Interfaces

An interface is a contract that an entity should conform to. Interfaces define properties, methods, and events, the members of an interface.

```
interface IPerson {  
    firstName:string,  
    lastName:string  
}
```

Interfaces can be passed to functions and functions can access the members.

```
function printFullName (person: IPerson) {  
    console.log(person.firstName);  
    console.log(person.lastName);  
}  
  
let myObject = { firstName: "Thayanithy", lastName: "Jegan" }  
printFullName(myObject);
```

Interfaces

1. The type-checker checks the call to `printFullName`. The `printFullName` function takes a single parameter that requires an object which has two properties called `firstName` and `lastName` of type `string`.
2. If the actual object passed to the function has more properties than this, but the compiler throws an error saying `Object literal may only specify known properties`.
3. The object passed to `printFullName` is not explicitly saying that it is implementing the interface `IPerson`. In TypeScript it is only the shape that matters. If the object we pass to the function meets the requirements listed, then it is allowed.

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. Interfaces with optional properties are written similar to other interfaces. The optional properties allows to describe these possibly available properties and also preventing use of properties that are not part of the interface.

```
interface IPerson {  
    fullName?: string;  
    icNumber?: string;  
}  
  
let newStudent:IPerson = {color: "white"};  
console.log(newStudent.firstName) => Throws error
```

Read only Properties

The readonly properties are modifiable only when an object is created.

```
interface Point {  
    readonly x: number;  
    readonly y: number;  
}
```

After assigning the object literal, x and y cannot be changed.

```
let p1: Point = { x: 10, y: 20 };  
p1.x = 5; // error!
```

TypeScript comes with a `ReadonlyArray<T>` type that is same as `Array<T>`

```
let a: number[] = [1, 2, 3, 4];  
let ro: ReadonlyArray<number> = a;  
ro[0] = 12; // error!  
ro.push(5); // error!
```

Interfaces and Classes

Interfaces can be passed around the functions

```
interface IPerson {  
    fullName:string,  
    sayHi: ()=>string  
}  
  
class Student implements IPerson {  
    fullName:string,  
    sayHi: ()=>string {  
        alert("Hello " + this.fullName);  
    }  
}
```

Interface and Objects

Object also can be derived from Interfaces

```
interface IPerson {  
    firstName:string,  
    lastName:string,  
    sayHi: ()=>string  
}  
  
var customer:IPerson {  
    firstName:"Tom",  
    lastName:"Hanks",  
    sayHi():string => { return "Hi, " + this.firstName + " " + this.lastName }  
}  
  
console.log("First Name : " + customer.firstName + " " + customer.lastName);  
console.log("Say Hi : " + customer.sayHi());
```

Interface Inheritance

An interface can be inherited from multiple interfaces.

```
interface Male { mustache:boolean }
```

```
interface Female { longhair:boolean }
```

```
interface Human extends Male, Female {  
}
```

```
var John:Human = { mustache:true, longhair:false }
```

```
console.log("John has mustache: "+John.mustache);
```

```
Console.log("John has long hair:"+John.longhair);
```

The object John is of type interface Human which inherited from Male and Female interfaces.

Modules

A module is designed with the idea to organize code written in TypeScript.

Modules are broadly divided into

1. **Internal Modules** are used to logically group classes, interfaces, functions into one unit and can be exported. This logical grouping is named as **namespace** in latest TypeScript.
2. **External Modules** exists to specify and load dependencies of multiple js files. Traditionally dependency management was done using browser script tags (<script>). But that's not extendable, as its linear while loading modules. Instead of loading files one after other there is no asynchronous option to load modules.

Internal Module Syntax

Internal Module Syntax (Old)

```
module Arithmetic {  
    export function add(x, y) {  
        console.log(x+y);  
    }  
}
```

Namespace Syntax (New)

```
namespace Arithmetic {  
    export function add(x, y) {  
        console.log(x + y);  
    }  
}
```

External Module

The syntax for declaring an external module is using keyword 'export' and 'import'.

Syntax

```
//FileName : SomeClass.ts  
export class SomeClass {  
    //code declarations  
}
```

To use the declared module in another file, an import keyword is used as given below. The file name is only specified no extension used.

```
import someClassRef = require("./SomeClass");
```


External Module

```
export class Circle {  
    public draw() {  
        console.log("Circle is drawn (external module)");  
    }  
}  
  
export class Triangle {  
    public draw() {  
        console.log("Triangle is drawn (external module)");  
    }  
}
```

External Module

```
import circle = require("./Circle");  
  
import triangle = require("./Triangle");  
  
function drawAllShapes(shapeToDraw: shape.IShape) {  
    shapeToDraw.draw();  
  
}  
  
drawAllShapes(new circle.Circle());  
  
drawAllShapes(new triangle.Triangle());
```

Advanced Type - Tuples

Declaring a Tuple

```
var mytuple = [10, "Hello"];
```

Accessing values in Tuple

```
console.log(mytuple[0]);
```

```
console.log(mytuple[1]);
```

Tuple Operations

```
mytuple.push(12)
```

```
mytuple.pop()
```

Updating Tuple

```
mytuple[0] = 121
```

Destructuring a Tuple

```
var [a, b, c] = mytuple
```

```
console.log(a);
```

Advanced Type – Union Type

TypeScript gives programs the ability to combine one or two types. Union types are a powerful way to express a value that can be one of the several types. Two or more data types are combined using the pipe symbol (`|`) to denote a Union Type.

```
var val:string|number;
```

```
val = 12;
```

```
console.log("Numeric value of val " + val);
```

```
val = "This is a string";
```

```
console.log("String value of val " + val);
```

INTRODUCTION TO ANGULAR

What is Angular

1. Angular is a popular **JavaScript framework or platform** that makes it easy to build web applications.
2. Angular brings object oriented web development to the mainstream, that is close to Java 8.
3. Angular empowers developers to build applications that live on the web, mobile, or the desktop.
4. Google even created a new language “AtScript” for Angular applications. But since Microsoft agreed to add support for decorators to TypeScript, TypeScript become the language for Angular framework.

Features

The main important features of Angular Framework are

1. Modules
2. Directives
3. Templates
4. Services
5. Component Based
6. Expressions
7. Two Way Data-Binding
8. Validations
9. Filters
10. Routing
11. Dependency Injection
12. Testing

Building Blocks

There are basically 8 building blocks of Angular and they are

- Components
- Data binding
- Templates
- Directives
- Modules
- Services
- Metadata
- Dependency Injection

Overview

Angular is not an MVC framework, it is component-based framework.

In Angular an application is a tree of loosely coupled **components**.

A sample e-Commerce

Webpage as a collection of

Components such as

1. Navigation Bar

2. Search

3. Carousel

4. Product



Overview - Components

In TypeScript a component is simply a class annotated with `@Component`:

```
@Component({  
  selector: 'auction-navbar',  
  template: ` HTML or other markup is in-lined here`  
})  
  
export default class HomeComponent {  
  // Application logic goes here  
}
```

1. `@Component` is used to define the component
2. The *selector* property identifies the name of this component
3. The *template* property is a placeholder for HTML markup

Overview - Template

Angular template consists of standard and custom HTML tags that represent respective components. The HTML can be in-lined or can be in a separate file. The property **templateURL** or **template** is used inside Component.



```
<auction-navbar></auction-navbar>

<div class="container">
  <div class="row">
    <div class="col-md-3">
      <auction-search></auction-search>
    </div>

    <div class="col-md-9">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>

<auction-footer></auction-footer>
```

Overview - Route

The e-Commerce page has three Product components. In Angular the rendering is done by binding the template to array of components retrieved from server. The title of each product is a link to associated product detail page. In a single page application (SPA) entire page doesn't get refreshed to display product details. The product details will be rendered in carousel area. This can be accomplished by

1. Angular's **router-outlet** directive, allows to declare area currently occupied by carousel and products, so that it can vary content based on user's navigation.
2. Encapsulate Carousel and Product inside the Home component
3. Create a new ProductDetail component
4. Configure the Angular's Router to show either the Home or ProductDetail component in the designated **<router-outlet>** area.

Overview - Services

Components use *services* for implementing business logic. Services are just classes that Angular instantiates and injects into components.

```
export class ProductService {  
    products: Product[] = [];  
    getProducts(): Array<Product> {  
        // The code to retrieve product  
        // into goes here return products;  
    }  
}
```

If the ProductService is declared inside the Component constructor, Angular will automatically instantiate and inject that service into the component.

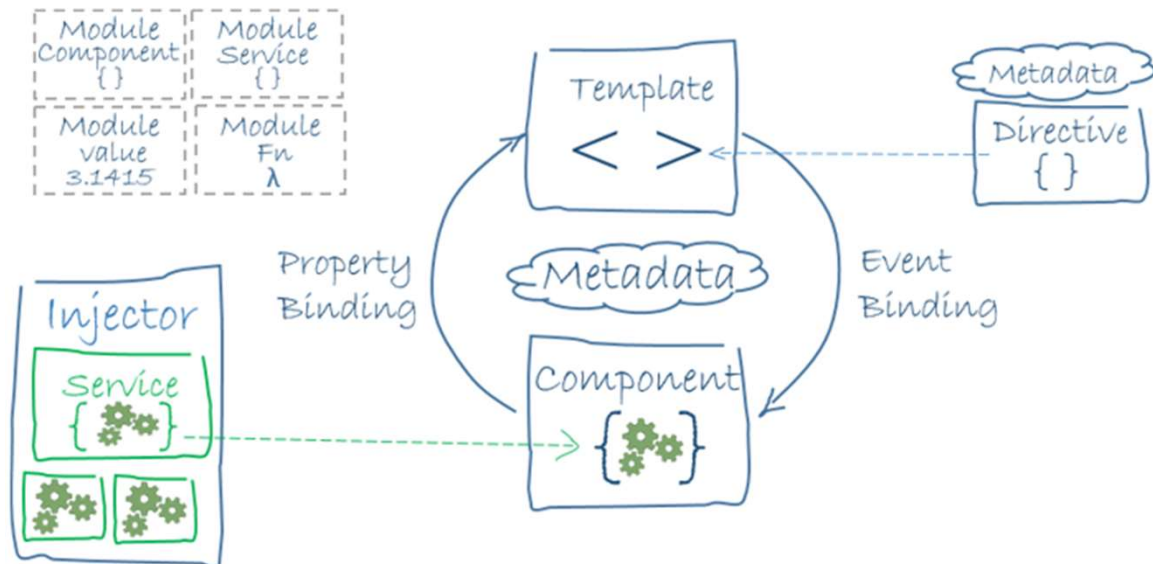
Overview - Services

```
@Component {  
    .....  
}  
  
export default class ECommerceComponent {  
    products: Product[] = [];  
    constructor(productService: ProductService) {  
        this.products = productService.getProducts();  
    }  
}
```

Angular dependency injection module is flexible, and it is easy to use because the objects can be injected via constructors.

Overview - Architecture

1. Angular applications are developed using HTML templates with angularized markup and component classes are to manage templates. Application logics are added in services. Components and services are boxed in modules. App is launched by bootstrapping the root module.
2. Angular takes over, presenting application in a browser and responding to user interactions.



Install Angular

1. Angular CLI is a command line interface for Angular. The Angular CLI makes it easy to create an Angular application that already works, right out of the box.
2. To install and use the Angular CLI command line interface as well as run the Angular application server we going to use npm (the Node.js package manager).
3. To install the Angular CLI go to command prompt and type
npm install -g @angular/cli
4. Now to create a new Angular application type
ng new HelloWorld

Install Angular

1. HelloWorld is the name of the folder for the application. The command creates the Angular application in TypeScript and install it's dependencies.
2. To run Angular application, navigate to the new folder and typing `ng serve` to start the web server and open the application in a browser.
cd HelloWorld
npm install
ng serve
3. You should see "Welcome to app!!" on *http://localhost:4200* in your browser.

Review project using VS Code

1. Open Angular application in VS Code, open another command prompt and navigate to the HelloWorld folder and type
cd HelloWorld
code .
2. Expand src\app folder and select the app.component.ts file.
 1. Syntax highlighting
 2. IntelliSense
 3. Go to Definition
 4. Peek Definition
3. Open app.component.ts and change the title from app to "Hello World".
Save the file, the running instance will update the page.

The Entry Point

The ***index.html*** is the entry point to the application. It

1. links to styles and javascript files
2. provides HTML markup for application with custom `<app-root>` element

The ***src/main.ts*** file has the bootstrapping logic. This file

1. import the `platformBrowserDynamic` object from the `'@angular/platform-browser-dynamic'` module
2. Calling the `bootstrapModule` function with the `AppModule` as argument tells Angular that this module is the main module for the application.



COMPONENTS



Introduction

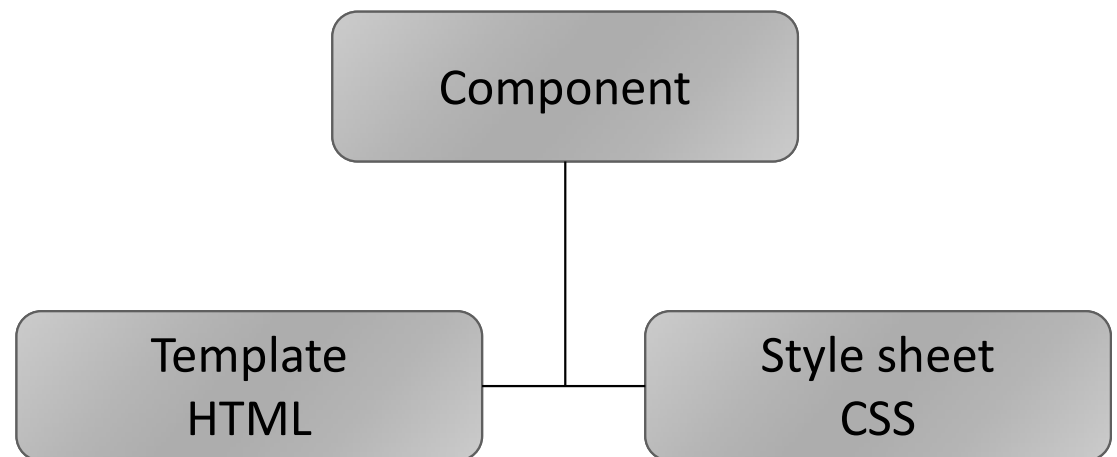
1. In Angular a component is basically a class that is used to show an element on the screen.
2. Components have properties that decide how they should look and behave on screen, which can be manipulate with the help of a component decorator.
3. Every component has a defined template which can communicate with code defined in the component class.
4. A component can be created, updated, destroy as user interact with the application. Life Cycle hooks are modules used for this purpose.

Component Decorator

Angular 4 components are simply classes that are designated as a Component with the help of a “*Component Decorator*”.



Every component has a defined template. The template can communicate with the code that has been defined in the class



View the Component Code

1. Open our HelloWorld Angular application in VS Code
cd HelloWorld
code .
2. Expand src\app folder and select the app.component.ts file.
3. Angular Component is structured into **3 different sections**

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

Component Import

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

1. To make a simple class as Angular component import the **Component** member from @angular/core library.
2. Components may have more than one import based on the needs of the component.
3. The project may have few services. If the component use any of those services through dependency injection, this is where those services are imported into the component.

Component Decorator

An angular component decorator, has a variety of configuration properties that help define the given component.

1. **selector**: This is the name of the tag that component is applied to. For instance: `<app-root>Loading...</app-root>` within index.html.
2. **templateUrl** & **styleUrls**: These define the HTML template and style sheets associated with this component. You can also use **template** and **styles** properties to define inline HTML and CSS.
3. **animations**: This is where animations are defined.
4. There are other properties which can be defined within the component decorator depending on the needs of the component.

Component Class

```
export class AppComponent {  
    title = 'app works!';  
}
```

1. Finally, the core of component, which is component class.
2. This is where the various properties and methods are defined.
3. Any properties and methods defined here are accessible from the template.
4. Events that occur in the template are accessible within the component.
5. This is where dependency injection occurs

Creating a Component

Components can be created using Angular CLI. In command prompt type

ng g component Feedback

1. **ng** invokes the Angular CLI
2. **g** is short for generate
3. **component** is the type of element going to be generated
4. **component name**

The ng command actually creates a new folder inside */app* folder

It generates CSS, HTML, TS (component class) and .spec.ts file for unit tests

It imports the newly generated component inside */src/app/app.module.ts*

Linking HTML Template

Now let us link the Feedback view to its parent component. Open the `app.component.html` file and add the Feedback Component selector element.

```
< app-feedback></ app-feedback>
```

Open the `feedback.component.html` file and add the following code

```
<div class="row">  
  <div class="col-xs-12">  
    <h1>Feedback</h1>  
    <p>We value our Guest Feedback</p><hr>  
  </div>  
</div>
```

LIFE CYCLE HOOKS

Introduction

A component has a lifecycle managed by Angular. Angular

1. creates the component
2. renders the component it
3. creates component children
4. renders component children
5. checks when data-bound properties change
6. destroys it before removing it from the DOM

Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.

Component Lifecycle Hooks

Each interface has a single hook method whose name is the interface name prefixed with ng. For example, OnInit has a hook method named `ngOnInit()`.

<code>ngOnChanges()</code>	Respond when Angular (re)sets data-bound input properties. Method receives a <code>SimpleChanges</code> object of current and previous property values. Called before <code>ngOnInit()</code> and when data-bound input properties change.
<code>ngOnInit()</code>	Initialize directive/component after Angular first displays data-bound properties and sets directive/component's input properties. Called once, after the first <code>ngOnChanges()</code> .
<code>ngDoCheck()</code>	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after <code>ngOnChanges()</code> and <code>ngOnInit()</code> .

Component Lifecycle Hooks

<code>ngAfterContentInit()</code>	Respond after Angular projects external content into the component's view. Called once after the first <code>ngDoCheck()</code> .
<code>ngAfterContentChecked()</code>	Respond after Angular checks the content projected into the component. Called after the <code>ngAfterContentInit()</code> and every subsequent <code>ngDoCheck()</code> .
<code>ngAfterViewInit()</code>	Respond after Angular initializes the component's views and child views. Called once after first <code>ngAfterContentChecked()</code> .
<code>ngOnDestroy</code>	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called just before Angular destroys the directive/component.

OnInit

Developers tap into key moments in lifecycle by implementing one or more of lifecycle hook interfaces in Angular core library.

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<ul><li *ngFor = "let lifehook of lifehooks">{{ lifehook }}</li></ul>`
})
export class AppComponent implements OnInit {
  lifehooks = ['Events'];
  constructor() { this.lifehooks.push('constructor'); }
  ngOnInit() {
    this.lifehooks.push('onInit');
  }
}
```

DoCheck

The `ngDoCheck()` is called during every change detection.

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

```
@Component({
```

```
  selector: 'app-root',
```

```
  template: `<div class="parent">
```

```
    <h2>DoCheck</h2><i>Enter a new Car</i><br>
```

```
    <input type="text" #newCar (keyup.enter)="addCar(newCar.value)" placeholder="Car">
```

```
    <button (click)="resetCars()">Reset</button><p><i>Click a Car to remove</i></p>
```

```
    <div *ngFor="let car of cars, let i=index" (click)="removeCar(i)">{{ car }}</div>
```

```
    <div *ngFor="let message of messages">{{ message }}</div>
```

```
  </div>`
```

```
  })
```

DoCheck

```
export class AppComponent implements DoCheck {  
  oldCars:string[];  
  cars:string[];  
  newCar:string;  
  messages:string[];  
  constructor() { this.resetCars(); }  
  ngDoCheck() {  
    if (this.oldCars.length !== this.cars.length) {  
      this.messages.push("Old Cars : " + this.oldCars + ",  
                          now new cars : " + this.cars);  
      this.oldCars = this.cars.slice();  
    }  
  }  
}
```

DoCheck

```
addCar(value: string) {  
    this.cars.push(value.trim());  
}  
removeCar(arrayindex: number) {  
    this.cars.splice(arrayindex, 1);  
}  
resetCars() {  
    this.oldCars = [];  
    this.cars = [];  
    this.newCar = "";  
    this.messages = [];  
}  
}
```



TEMPLATE



External & Inline Templates

External HTML Templates: Use property **templateUrl** to define a HTML template when there is considerable amount of HTML associated with given component.

```
@Component({  
    selector: 'app-root',  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
})
```

Inline HTML Templates: Use property **template** to define the single line of HTML.

```
template: '<h1>Hey guys!</h1>',
```

To define multiple lines of inline HTML, use backticks ``.

```
template: `<h1>Hey guys!</h1>  
<p>How are you dong?</p>`,
```

Display Data - Interpolation

Easiest way to display a component property is to bind property name through

interpolation. Open app.component.ts file and change template as follows

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
              <h2>My favorite car is: {{myCar}}</h2> `
})
export class AppComponent {
  title = 'Tour of Cars';
  myCar = 'Proton Saga';
}
```

Two variables title and myCar are declared and displayed using interpolation.

Display Data - Interpolation

1. Angular automatically pulls the value of title and myCar properties from the component and inserts those values into the browser.
2. Angular updates display when these properties change or after some kind of asynchronous event related to the view, such as a keystroke, a timer completion, or a response to an HTTP request.

Interpolation pull data without creating an instance of AppComponent class.

CSS selector in @Component decorator specifies an element named <my-app>, a placeholder in index.html. During bootstrap with AppComponent in main.ts, Angular looks for <my-app> in index.html, finds it, instantiates an instance of AppComponent, and renders it.

Display Array - Interpolation

To display list of cars, begin by adding an array of car names to component.

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `<ul><li *ngFor="let car of cars"> {{ car }} </li></ul> `
})
export class AppComponent {
  cars = ['Proton', 'Volvo', 'Toyota', 'Mercedes'];
}
```

UI uses HTML unordered list with `` and `` tags. The `*ngFor` in `` element is Angular **repeater directive**. It marks that `` element (and its children) as the "repeater template"

Display Class - Interpolation

Create a new file in app folder called car.ts with the following code:

```
export class Car {  
    constructor( public id: number, public name: string) { }  
}
```

After importing the Car class, the AppComponent.cars property can return a *typed* array of Car objects:

```
cars = [  
    new Car(1, 'Wira'),  
    new Car(13, 'Saga'),  
    new Car(15, 'Myvi'),  
    new Car(20, 'Alza')  
];
```

Display Class - Interpolation

Now update the template. At the moment the current template displays the car's id and car's name. Fix that to display only the car's name property.

```
template: `<h1>{{title}}</h1>
  <p>Cars:</p>
  <ul>
    <li *ngFor="let car of cars"> {{ car.name }} </li>
  </ul> `
```

Prohibited Operators

JavaScript expressions which are **prohibited** in angular interpolation are

1. assignments (=, +=, -=, ...)
2. new
3. chaining expressions with ; or ,
4. increment and decrement operators (++ and --)
5. no support for the bitwise operators | and &

Property Binding

Property binding sets an element property to a component property value.

An example is binding the src property of an image element.

```
export class AppComponent {  
    title = 'app';  
    imageUrl = 'assets/screen.jpg';  
    titleClass = 'red-title';  
}
```

```
<img [src]="imageUrl">
```

```
<h1 [class]="titleClass">HELLO</h1>
```

```
.red-title {  
    color:red;  
}
```

Property Binding

There are different syntax for property binding are

1. If the value is string then property binding can be done through Interpolation

```

```

2. If the value is not string then have to use the following syntax

```
<img [src]="imageUrl" style="height:30px">
```

```

```

3. The common usage of property binding is to enable or disable an input field based on condition

```
<button [disabled]="buttonStatus">Submit</button>
```

```
<button [disabled]="buttonStatus == 'NoClick'">Submit</button>
```

Attribute Binding

Differences between HTML attribute and DOM property

1. Attributes are defined by HTML and properties are defined by DOM
2. DOM properties can change but HTML attributes cannot
3. HTML attributes have corresponding DOM property and DOM properties have corresponding HTML attributes - **id**
4. HTML attributes that do not have DOM property - **colspan**
5. DOM properties that do not have HTML attributes - **textContent**
6. HTML attribute value contains initial value where as DOM property value contains current value.

Attribute binding syntax start with the prefix **attr**, followed by a dot (.)

```
<td [attr.colspan]="1 + 1">One-Two</td>
```

Class Binding

Sometimes the developer needs to change the CSS on the fly.

1. Class binding allows to add, remove and change CSS classes from an element's class attribute based on component property.
2. Class binding syntax start with the prefix class followed by a dot (.) and name of a CSS class [class.class-name]

Code to add and remove CSS class

```
<div [class.titleClass]="isSpecial">
```

The class binding is special

```
</div>
```

```
<h1 [class]="titleClass">HELLO</h1>
```


Style Binding

Style binding syntax start with prefix style, followed by a dot (.) and the name of a CSS style property [style.style-property]

```
<button [style.color]="isSpecial ? 'red': 'green'">Red</button>
```

```
<button [style.background-color]="canSave ? 'cyan': 'grey'">Save</button>
```

Some style binding styles have a unit extension. The following example conditionally sets the font size in “em” and “%” units .

```
<button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>
```

```
<button [style.font-size.%]="!isSpecial ? 150 : 50" >Small</button>
```

Style property name can be written in either dash-case or camelCase, such as font-size or fontSize.

NgClass

NgClass directive allows to set CSS class dynamically for a DOM element

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

```
<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>
```

NgStyle

NgStyle directive lets to set a given DOM elements style properties.

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

This sets the background color of the div to green.

```
<h4>NgStyle</h4>
```

```
<ul *ngFor="let person of people">
```

```
    <li [ngStyle]='{'color':getColor(person.country)}'>
```

```
        {{ person.name }} ({{ person.country }})
```

```
    </li>
```

```
</ul>
```

NgStyle

The method **getColor**

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

Summary

Interpolation can be

1. in between the HTML element tags

```
<h3> {{title}} </h3>
```

2. within the attribute assignments

```

```

```
<img [src]="heroImageUrl" style="height:30px">
```

```
<img [attr.src]="heroImageUrl" style="height:30px">
```

3. expression

```
<p>The sum of 1 + 1 is {{1 + 1}}</p>
```

4. expression can invoke methods of the host component

```
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
```

EVENT BINDING

Events

1. Angular Event Bindings are used to capture user-initiated events from the view and calling the corresponding component logic.
2. Angular Event bindings are used to respond to any DOM Events raised by user actions such as clicking a link, pushing a button and keyboard action such as entering text.
3. Angular Event bindings are one way data binding that sends information from the view to the component class which is opposite to property binding where the data is send from component class to the view.

Binding to User Input Events

To bind to a DOM event, surround DOM event name in parentheses and assign a quoted template statement to it.

```
<button (click)="onClickMe()">Click me!</button>
```

```
<button (click)="onClickMe($event)">Click me!</button>
```

1. **(click)** identifies button's click event as **target of binding**.
2. **Text in quotes** is template statement. The template statement is the component's method which responds to that event.

The following URL offers a list of events that defined by browsers

<https://developer.mozilla.org/en-US/docs/Web/Events>

Events

Identifiers in a template statement belong to the Angular component. The code is HTML that has a button attached with `onClickMe` method.

```
@Component({  
    selector: 'click-me',  
    template: `<button (click)="onClickMe()">Click me!</button>  
                {{message}}`  
})  
export class ClickMeComponent {  
    message = "";  
    onClickMe() {  
        this.message = 'You are my hero!';  
    }  
}
```

Object \$event

DOM events carry a payload of information that may be useful to the component. The following code listens to keyup event and passes entire event payload (\$event) to the component event handler.

```
template: `<button (click)="onClickMe($event)">Submit</button>`  
export class EventDemoComponent {  
    onClickMe(event: any) {  
        console.log(event);  
    }  
}
```

The properties of an \$event object vary depending on the type of DOM event. For example, a mouse event includes different information than a keyboard event.

Get User Input

The following code listens to the keyup event and passes the entire event payload (\$event) to the component event handler.

```
template: `<input (keyup)="onKey($event)"> <p>{{values}}</p>`  
export class GetUserInputComponent {  
    values = "";  
    onKey(event: any) {  
        this.values += event.target.value + ' | '  
    }  
}
```

DOM event objects have a target property, a reference to element that raised event. In this case, target refers to <input> element and the event.target.value returns current contents of that element.

Get User Input

Casting the `$event` with any type simplifies code but at a cost. Event can be declared as `KeyboardEvent`.

```
export class GetUserInputComponent {  
    values = '';  
    onKey(event: KeyboardEvent) {  
        this.values += (<HTMLInputElement>event.target).value + ' | ';  
    }  
}
```

Not all elements have value property so it casts target to an input element.

Now since the event object is typed with `KeyboardEvent`, it reveals a significant objection to passing the entire DOM event into the method.

Using template reference variable

Template reference variables provide direct access to an element from within the template. To declare a template reference variable, precede an identifier with a hash (or pound) character (#).

```
@Component({  
  selector: 'key-up',  
  template: `<input #box (keyup)="0">  
    <p>{{box.value}}</p>`  
})
```

The code uses box variable to get input element's value and display it with interpolation between <p> tags. The template doesn't bind to the component and the component does nothing.

Using template reference variable

Let us rewrite the previous keyup example to get the user's input.

```
@Component({
  selector: 'key-up',
  template: `<input #box (keyup)="onKey(box.value)">
    <p>{{values}}</p> `
})
export class KeyUpComponent_v2 {
  values = "";
  onKey(value: string) {
    this.values += value + ' | ';
  }
}
```

This approach makes the component gets clean data values from the view. It no longer requires knowledge of the \$event and its structure.

Key event filtering

The (keyup) event handler hears every keystroke. Sometimes only the Enter key matters, because it signals that the user has finished typing. The code can examine every `$event.keyCode` and take action.

```
@Component({
  selector: 'key-up3',
  template: `<input #box (keyup.enter)="onEnter(box.value)">
    <p>{{value}}</p> `
})
export class KeyUpComponent_v3 {
  value = "";
  onEnter(value: string) {
    this.value = value;
  }
}
```

On Blur

Current state of input box is lost if the user moves away or clicks elsewhere on the page without pressing Enter. The component's value property is also updated when the user presses Enter.

```
@Component({
  selector: 'key-up',
  template: `<input #box (keyup.enter)="update(box.value)"
    (blur)="update(box.value)"><p>{{value}}</p>`
})
export class KeyUpComponent_v4 {
  value = "";
  update(value: string) {
    this.value = value;
  }
}
```


Put it all together

User can add a car by typing car's name in input box and click Add.

```
@Component({
  selector: 'car-manager',
  template: `<input #newCar (keyup.enter)="addCar(newCar.value)"
    (blur)="addCar(newCar.value); newCar.value=""">
    <button (click)="addCar(newCar.value)">Add</button>
    <ul><li *ngFor="let car of cars">{{car}}</li></ul>`
})
export class CarComponent {
  cars = ['Proton', 'Volvo', 'Mazda', 'Toyota'];
  addCar(newCar: string) {
    if (newCar) this.cars.push(newCar);
  }
}
```

Two Way Data Binding

3 types of data bindings in Angular are Interpolation, Property Binding and Event Binding. All are one way data binding. Two way Data Binding is achieved using Property Binding and Event Binding together.

```
<input [value]="name" (input)="name=$event.target.value" />
```

```
<h1>Hello {{name}} </h1>
```

1. `[value]="name"` is property binding. We are binding value property of input element with variable (or expression) name.
2. `(input)="expression"` is event binding. Whenever input event occur expression will be executed. `"name=$event.target.value"` is an expression which assigns entered value to name variable.

Two Way Data Binding

Angular does not have any built in two-way data binding. However, the ngModel directive implements two-way data binding. It comes with a property and event binding

```
<input [(ngModel)]="username">
```

```
<input [ngModel]="username" (ngModelChange)="username=$event">
```

```
<p>Hello {{username}}!</p>
```

Property binding [ngModel] takes care of updating DOM element.

1. Event binding (ngModelChange) notifies when there was a change in the DOM.

Using ngModel

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

@Component({
  selector: 'example-app',
  template: `
    <input [(ngModel)]="name" #ctrl="ngModel" required>
    <p>Value: {{ name }}</p>
    <p>Valid: {{ ctrl.valid }}</p>
    <button (click)="setValue()">Set value</button>`,
})
export class SimpleNgModelComp {
  name: string = '';
  setValue() {
    console.log(this.name);
    this.name = 'Nancy';
  }
}
```

Using ngModel

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

@Component({
  selector: 'example-app',
  template: `<input [ngModel]="name"
    (ngModelChange)="onModelChange(oldVal, $event);
    oldVal = $event;">
    <p>Value: {{ oldVal }}</p>`,
})
export class SimpleNgModelComp {
  name: string = '';
  onModelChange(event) {
    if (this.name !== event) console.log(event);
  }
}
```



COMPONENT STYLES



Introduction

For every Angular component developers will define a HTML template and also CSS styles that go with the template.

1. Angular applications are styled with standard CSS.
2. Angular bundle *component styles* with components, enabling modular design than regular style sheets.

There are two ways to set CSS to any Angular components

1. One way to do this is by setting styles property in the component Decorator. Styles property takes an array of CSS code strings.
2. Another way to do this is to load styles from external CSS files by adding a styleUrls attribute into component decorator.

Setting the styles

Set the styles property in the component metadata

```
@Component({  
    selector: 'hero-app',  
    template: `<h1>Tour of Heroes</h1>  
    <hero-app-main [hero]=hero></hero-app-main>`,  
    styles: ['h1 { font-weight: normal; }']  
})  
  
export class HeroAppComponent { /* ... */ }
```

1. CSS put in the styles apply only within the template of that component.
2. The CSS does not collide with classes and selectors used elsewhere in the application.
3. Similarly changes to styles elsewhere in the application don't affect the component's styles.

Special Selectors

:host

The :host pseudo-class selector to target styles in the element that hosts the component.

```
:host {  
    display: block;  
    border: 1px solid black;  
}
```

Use the function form to apply host styles conditionally by including another selector inside parentheses after :host.

```
:host(.active) {  
    border-width: 3px;  
}
```

Special Selectors

:host-context

The :host-context pseudo-class selector to target styles in the element that hosts the component.

```
:host-context {  
    display: block;  
    border: 1px solid black;  
}
```

Use the function form to apply host styles conditionally by including another selector inside parentheses after :host.

```
:host(.active) {  
    border-width: 3px;  
}
```

Loading Component Styles

There are several ways to add styles to a component:

1. By setting styles

```
styles: ['h1 { font-weight: normal; }']
```

2. By Setting styleUrls metadata

```
styleUrls: ['app/hero-details.component.css']
```

3. Inline in the template HTML.

```
template: ` <style>h3 { background-color: red; }</style>  
<h3>Controls</h3> `
```

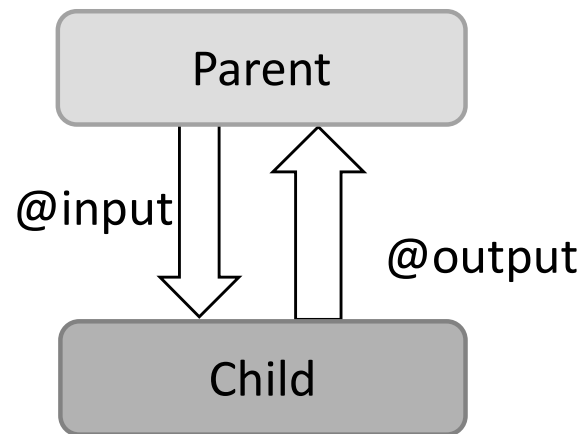
4. Template Link tags

```
template: ` <link rel="stylesheet" href="app/hero-team.component.css">  
<h3>Team</h3> `
```

COMPONENT INTERACTION

Communicating

Components can share data or communicate among themselves using `@input` and `@output` decorators



First we going to see how the child get data from the parent using the `@input` decorator

Pass data with Input Binding

Parent component pass data to child component using “input binding”.

Step 1: Let us create a class called car.ts

```
export class Car {  
    name: string;  
}  
  
export const CARS = [  
    {name: 'Proton SAGA'},  
    {name: 'Proton WIRA'},  
    {name: 'Proton ISWARA'},  
    {name: 'Proton PERDANA'},  
    {name: 'Proton SATRIA'},  
    {name: 'Proton WAJA'},  
    {name: 'Proton GEN 2'},  
    {name: 'Proton PERSONA'}  
];
```

Pass data with Input Binding

Step 2: Let us create Child Component “CarChildComponent” with two input properties, typically adorned with @Input decorations.

```
import { Component, Input } from '@angular/core';
import { Car } from './car';
@Component({
  selector: 'car-child',
  template: `<h3>{{car.name}} says:</h3>
    <p>I, {{car.name}}, am at your service, {{masterName}}.</p>`
})
export class CarChildComponent {
  @Input() car: Car;
  @Input('master') masterName: string;
}
```

Second @Input aliases masterName as 'master'

Pass data with Input Binding

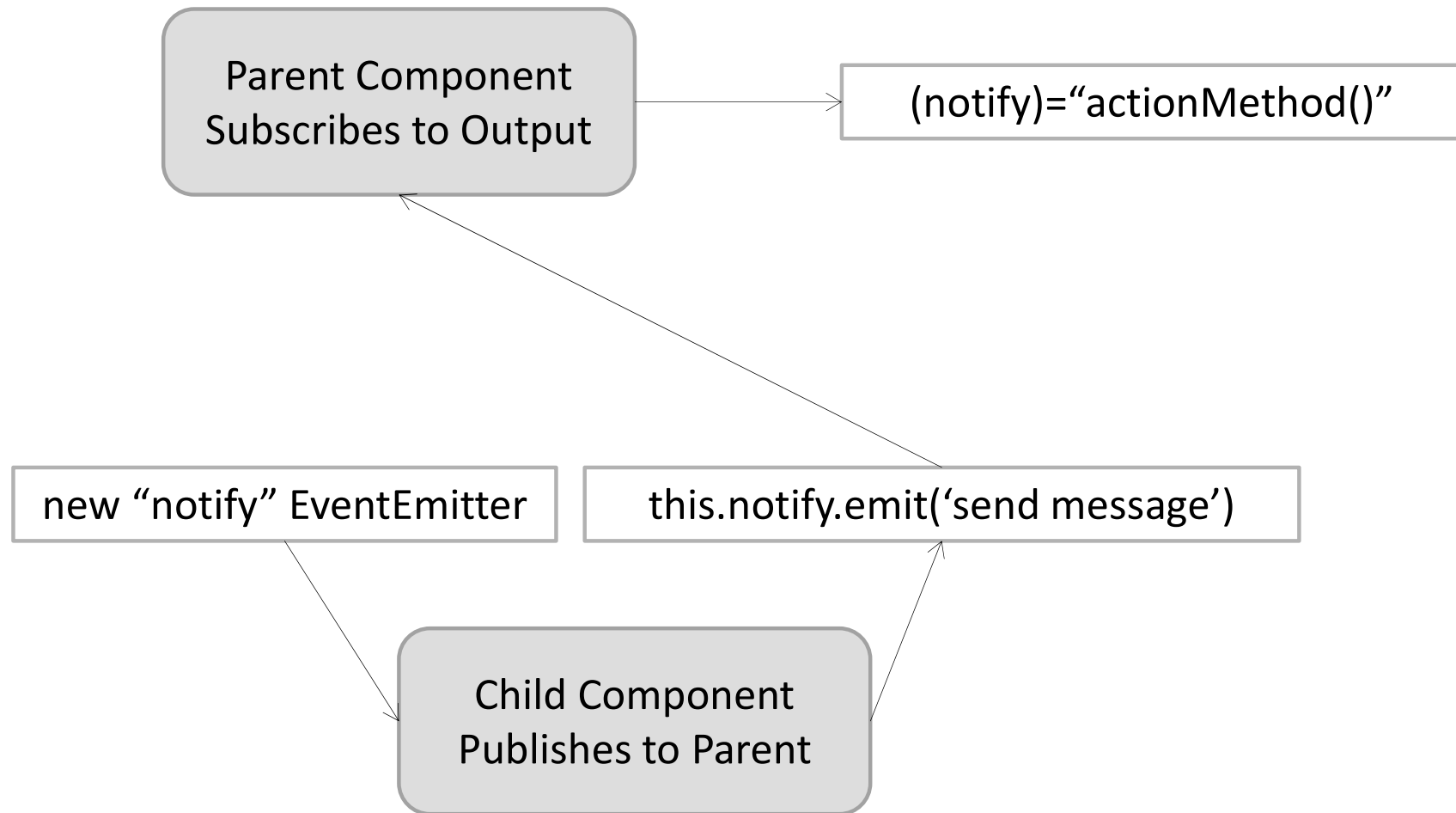
Step 3: Let us create “CarParentComponent” nests child CarChildComponent inside an *ngFor repeater, binding master string property to child's master alias, and each iteration's car instance to child's car property.

```
import { Component } from '@angular/core';
import { CARS } from './hero';
@Component({
  selector: 'car-parent',
  template: `<h2>{{master}} controls {{cars.length}} Cars</h2>
    <car-child *ngFor="let car of cars" [car]="car" [master]="master"></car-child>`
})
export class HeroParentComponent {
  cars = CARS;
  master = 'Master';
}
```


Pass data with Output Binding

1. `@Output` are used to create “custom events” that pass data from component to the outside world.
2. Share some information from nested/child component to parent component and display that information in parent component.
3. Create a property and assign to `@Output`. Similar to creating an Input property. The difference is we use `EventEmitter`.
`@Output() notify: EventEmitter<string> = new EventEmitter<string>();`
4. `EventEmitter` is an object that listens for something to happen and emits an event when triggered.
5. This is what know as observer pattern. The parent subscribe to the event emitted by the child.

Pass data with Output Binding



Pass data with Output Binding

Open the Child Component File

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

@Component({
  selector: 'car-child',
  template: `<div><button (click)="onClick()">Click Me</button></div>`
})

export class CarChildComponent {
  @Output() notify: EventEmitter<string> = new EventEmitter<string>();
  onClick():void {
    this.notify.emit('Message from child');
  }
}
```

Pass data with Output Binding

Open the Parent Component File

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

@Component({
  selector: 'app-root',
  template: `<h2>Parent</h2>From Child : {{ showMessage }}
    <table border="1" width="100%"><tr><td>
    <car-child (notify)="onNotifyClicked($event)"></car-child>
    </td></tr></table>`
})

export class AppComponent {
  showMessage:string = "";
  onNotifyClicked(message:string):void { this.showMessage = message; }
}
```

STRUCTURAL DIRECTIVES

Introduction

1. Structural directives are responsible for HTML layout. They shape or reshape the DOM's *structure*, typically by adding, removing, or manipulating elements.
2. Structural directives are easy to recognize. An asterisk (*) precedes the directive attribute name as in this example.

```
<div *ngIf="hero" >{{hero.name}}</div>
```

3. No brackets. No parentheses. Just *ngIf set to a string.
4. Angular changes this notation into a marked-up <ng-template> that surrounds the host element and its descendants. Each structural directive does something different with that template.

Types

Three common built-in structural directives are

1. NgIf

```
<div *ngIf="hero" >{{hero.name}}</div>
```

2. NgFor

```
<li *ngFor="let hero of heroes">{{hero.name}}</li>
```

3. NgSwitch

```
<ul [ngSwitch]="lang">
```

```
  <li *ngSwitchCase="'ms'">Malay</li>
```

```
  <li *ngSwitchCase="'ta'">Tamil</li>
```

```
  <li *ngSwitchDefault>English</li>
```

```
</ul>
```

NgIf

NgIf is the simplest structural directive and easiest to understand. It takes a boolean expression and makes a chunk of DOM appear or disappear.

```
<p *ngIf="true">
```

This paragraph is in the DOM.

```
</p>
```

```
<p *ngIf="false">
```

This paragraph is not in the DOM.

```
</p>
```

ngIf directive does not hide elements with CSS. It adds and removes them physically from the DOM.

NgIf – How it works

```
<div *ngIf="hero" >{{hero.name}}</div>
```

Angular translates `*ngIf="..."` into a template attribute.

```
<div template="ngIf hero">{{hero.name}}</div>
```

Then it translates template attribute into a `<ng-template>` element.

```
<ng-template [ngIf]="hero">
```

```
<div>{{hero.name}}</div>
```

```
</ng-template>
```

The `*ngIf` directive moved to `<ng-template>` element where it became a property binding `[ngIf]`. The rest of the `<div>` including its class attribute, moved inside the `<ng-template>` element.

NgSwitch

Angular *NgSwitch* is actually a set of cooperating directives such as *NgSwitch*, *NgSwitchCase* and *NgSwitchDefault*.

```
<ul [ngSwitch]="lang">  
  <li *ngSwitchCase="ms">Malay</li>  
  <li *ngSwitchCase="ta">Tamil</li>  
  <li *ngSwitchDefault>English</li>  
</ul>
```

1. *NgSwitch* is not a structural directive. It is *attribute* directive that controls behaviour of other two switch directives. That's why its not **ngSwitch*.
2. *NgSwitchCase* and *NgSwitchDefault* are structural directives.

The <ng-template>

<ng-template> is an Angular element for rendering HTML. It is never displayed directly. Before rendering, Angular *replaces* <ng-template> and its contents with a comment.

1. If some elements are wrap in a <ng-template> and there is no structural directive those elements disappear.

```
<p>Hip!</p>
```

```
<ng-template><p>Hip!</p> </ng-template>
```

```
<p>Hooray!</p>
```

2. Angular erases the middle "Hip!".
3. A structural directive puts a <ng-template> to work.

The <ng-container>

1. Structural directives are hosted by a root element. The list element () is a typical host element of an NgFor repeater.
`<li *ngFor="let hero of heroes">{{hero.name}}`
2. Native HTML container element, such as <div> also can be root element.
`<div *ngIf="hero" >{{hero.name}}</div>`

First problem: Sometimes the grouping element may break the template appearance because of CSS styles declared elsewhere.

```
<p> I turned the corner  
<span *ngIf="hero"> and saw {{hero.name}}. I waved </span>  
and continued on my way. </p>
```

Lets say there is a CSS style

```
p span { color: red; font-size: 70%; }
```

I turned the corner **and saw Mr. Nice. I waved** and continued on my way.

The <ng-container>

Second problem: Some HTML elements require all immediate children to be of a specific type. For example, <select> element requires <option> children. The options in a conditional <div> or a .

```
<select [(ngModel)]="hero">
```

```
  <span *ngFor="let h of heroes">
```

```
    <span *ngIf="showSad || h.emotion !== 'sad'">
```

```
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
```

```
    </span>
```

```
  </span>
```

```
</select>
```

The <ng-container>

1. Angular <ng-container> is a grouping element that doesn't interfere with styles or layout because Angular doesn't put it in the DOM.
2. The same conditional paragraph using <ng-container>.

<p>

I turned the corner

<ng-container *ngIf="hero">and saw {{hero.name}}. I waved</ng-container>

and continued on my way.

</p>

3. Now the content renders properly
4. <ng-container> is a syntax element recognized by Angular parser. It's not a directive, component, class, or interface. It's like curly braces.

ngClass – user.component.css

ngClass is used to add and remove CSS classes on an HTML element. Several CSS classes can be bind to ngClass using string, array, object.

```
.one {  
    color: green;  
}  
.two {  
    font-size: 20px;  
}  
.three {  
    color: red;  
}  
.four {  
    font-size: 15px;  
}
```

ngClass – user.component.html

```
<p [ngClass]="one two">Using NgClass with String. </p><br/>
```

```
<p [ngClass]="['three', 'four']"> Using NgClass with Array. </p><br/>
```

```
<p [ngClass]="{'one': true, 'three': false }">Using NgClass with Object.</p>
```

```
<div *ngFor="let user of users; let flag = even;">
```

```
    <div [ngClass]="{'one':flag, 'two':flag, 'three':!flag, 'four':!flag}">
```

```
        {{user}}
```

```
    </div>
```

```
</div><br/>
```

```
<div [ngClass]="getCSSClasses('nightMode')">
```

```
    Using NgClass with Component Method.
```

```
</div>
```


ngClass – user.component.ts

```
import {Component} from '@angular/core';
@Component({
  selector: 'user-app',
  templateUrl: 'app/user.component.html',
  styleUrls: ['app/user.component.css']
})
export class UserComponent {
  users = [ 'Mahesh', 'Krishna', 'Narendra', 'Jitendra' ];
  getCSSClasses(flag:string) {
    let cssClasses;
    if(flag == 'nightMode') { cssClasses = { 'one': true, 'two': true }
    } else { cssClasses = { 'two': true, 'four': false }
    }
    return cssClasses;
  }
}
```

Write a structural directive

Write an UnlessDirective structural directive that is opposite of NgIf. NgIf displays content when condition is true. This will displays content when condition is false.

`<p *appUnless="condition">Show this sentence unless the condition is true.</p>`

Creating a directive is similar to creating a component.

1. Import the Directive decorator.
2. Import the Input, TemplateRef and ViewContainerRef symbols; you'll need them for *any* structural directive.
3. Apply the decorator to the directive class.
4. Set the CSS *attribute selector* that identifies the directive when applied to an element in a template.

Create a directive using the command **ng g directive Unless**

Write a structural directive

Initial Directive Code

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';  
  
@Directive({ selector: '[appUnless]' })  
  
export class UnlessDirective {  
  
}
```

1. The directive's *selector* is in square brackets [myUnless]. The bracket define a CSS attribute selector.
2. The directive *attribute name* should be spelled in *lowerCamelCase* and begin with a prefix. In this example, the prefix is app.
3. The directive *class* name ends with the word Directive.

TemplateRef & ViewContainerRef

A simple structural directive like this one creates an embedded view from the Angular-generated `<ng-template>` and inserts that view in a view container adjacent to the directive's original `<p>` host element.

1. `<ng-template>` contents can be acquire with a Template Ref
2. The *view container* can be access through ViewContainerRef

Both element can be injected inside the directive constructor as private variables of the class.

```
constructor (private templateRef: TemplateRef<any>,  
             private viewContainerRef: ViewContainerRef) {  
  
}
```

The appUnless property

The directive consumer expects to bind a true/false condition to [appUnless]. That means the directive needs a appUnless property, decorated with @Input

```
@Input() set appUnless(condition: boolean) {  
    if (!condition) {  
        this.viewContainerRef.createEmbeddedView(this.templateRef);  
    } else {  
        this.viewContainerRef.clear();  
    }  
}
```

The appUnless property

Angular sets the appUnless property whenever the value of the condition changes.

1. If the condition is false and the view hasn't been created previously, tell the view container to create the embedded view from the template.
2. If the condition is true and the view is currently displayed, clear the container which also destroys the view.
3. Nobody reads appUnless property so it doesn't need a getter.

The complete code

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({ selector: '[appUnless]' })

export class UnlessDirective {

    private hasView:boolean = false;

    constructor(private templateRef: TemplateRef<any>,
                private viewContainer: ViewContainerRef) {

    }
```

The complete code

```
@Input() set appUnless(condition: boolean) {  
    if (!condition && !this.hasView) {  
        this.viewContainer.createEmbeddedView (this.templateRef);  
        this.hasView = true;  
    } else if (condition && this.hasView) {  
        this.viewContainer.clear();  
        this.hasView = false;  
    }  
}  
}
```


The complete code

1. Add this directive to the declarations array of the AppModule.
2. Then create some HTML to try it.

```
<p *appUnless="condition" class="unless a">
```

```
This is paragraph (A)
```

```
</p>
```

```
<p *appUnless="!condition" class="unless b">
```

```
This is paragraph (B)
```

```
</p>
```

3. When condition is false, top (A) paragraph appears and bottom (B) paragraph disappears. When condition is true, top (A) paragraph is removed and bottom (B) paragraph appears.

ATTRIBUTE DIRECTIVES

Attribute Directives - Create

Attribute directives are used as attributes of elements. Attribute directive changes the appearance or behaviour of a DOM element.

Create the directive using **ng g directive Highlight**

```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({
    selector: '[appHighlight]'
})

export class HighlightDirective {
    constructor(el: ElementRef) {
        el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

Attribute Directives - Create

1. The import statement specifies symbols from Angular core
 - Directive provides the functionality of the @Directive decorator
 - ElementRef injects into the directive's constructor so the code can access the DOM element.
 - Input allows data to flow from the binding expression into the directive.
2. The @Directive decorator function contains the directive metadata in a configuration object as an argument.
3. @Directive requires a CSS selector to identify the HTML in the template that is associated with the directive.

Apply the Attribute Directive

Open template app.component.html file and add the following code

```
<p appHighlight>Highlight me!</p>
```

Add an import statement to fetch the Highlight directive and add that class to the declarations NgModule metadata.

```
import { HighlightDirective } from './highlight.directive';
```

```
@NgModule({
```

```
  declarations: [
```

```
    AppComponent,
```

```
    HighlightDirective
```

```
  ],
```

```
})
```

Respond to User - HostListener

1. Currently, appHighlight simply sets an element color. Let us make it more dynamic by adding user events. Begin by adding HostListener and Input symbol to list of imported symbols

```
import { Directive, ElementRef, HostListener, Input } from  
'@angular/core';
```

2. Add two eventhandlers that respond when the mouse enters or leaves
- ```
@HostListener('mouseenter') onMouseEnter() { this.highlight('yellow'); }
@HostListener('mouseleave') onMouseLeave() { this.highlight(null); }
private highlight(color: string) {
 this.el.nativeElement.style.backgroundColor = color;
}
```

# Pass values into the Directive

---

Currently highlight colour is hard-coded within directive. That is inflexible.

Start by adding a highlightColor property to directive class.

```
@Input() highlightColor: string;
```

The @Input decorator adds metadata to class that makes the directive's highlightColor property available for binding. It is called an input property because data flows from the binding expression into the directive. Add the following into app.component.html

```
<p appHighlight highlightColor="yellow">Highlighted in yellow</p>
```

```
<p appHighlight [highlightColor]="orange">Highlighted in orange</p>
```

# Pass values into the Directive

---

It would be nice to *simultaneously* apply the directive and set the colour *in the same attribute* like this.

```
<p [appHighlight]="color">Highlight me!</p>
```

Name the directive property whatever you want *and **alias it*** for binding purposes.

```
@Input('appHighlight') highlightColor: string;
```

Modify the onMouseEnter() method to use it

```
@HostListener('mouseenter') onMouseEnter() {
 this.highlight(this.highlightColor || 'red');
}
```



# Update the Component

---

Update app.component.html as follows:

```
<h1>My First Attribute Directive</h1>
<h4>Pick a highlight color</h4>
<div>
 <input type="radio" name="colors" (click)="color='lightgreen'">Green
 <input type="radio" name="colors" (click)="color='yellow'">Yellow
 <input type="radio" name="colors" (click)="color='cyan'">Cyan
</div>
<p [myHighlight]="color">Highlight me!</p>
```

Revise the AppComponent.color so that it has no initial value.

```
export class AppComponent {
 color: string;
}
```

# SERVICES & DEPENDENCY INJECTION

# Dependency Injection

---

**Definition:** Dependency Injection is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself. If there is no DI the car class will be

```
export class Car {
 public engine: Engine;
 public tires: Tires;
 constructor() {
 this.engine = new Engine();
 this.tires = new Tires();
 }
 drive() {
 return `${this.engine.cylinders} cylinders ${this.tires.make} tires.`;
 }
}
```

# Dependency Injection

---

Since Car class creates Engine and Tires inside constructor, Car class is brittle, inflexible and hard to test. Car needs an engine and tires and Car constructor instantiates its own copies from the specific classes Engine and Tires. The problem is

1. If Engine class evolves and its constructor requires a parameter, that would break Car class and Car code has to be rewritten as follows

*this.engine = new Engine(theNewParameter)*

Initially Engine constructor parameter was not even in the plan. But now have to start worrying about it because when definition of Engine changes, Car class must change. That makes Car **brittle**.

2. If Car needs a different brand of tires, cannot do anything because Car class is locked with specific brand. That makes Car class **inflexible**.

# Dependency Injection

---

To make the Car more robust, flexible, and testable change the Car constructor to a version with DI:

```
export class Car {
 constructor(public engine: Engine, public tires: Tires) {
 }
 drive() {
 return `${this.engine.cylinders} cylinders ${this.tires.make} tires.`;
 }
}
```

The definition of dependencies are now as parameters in constructor. Car class no longer creates an engine or tires. It just consumes them.

# Dependency Injection

---

Now create a car by passing engine and tires to the constructor.

```
let car = new Car(new Engine(), new Tires());
```

Definition of engine and tire dependencies are decoupled from Car class. If someone extends Engine class, that is not Car's problem.

```
class PowerEngine {
 constructor(public cylinders: number) { }
}
```

```
let bigCylinders = 12;
```

```
let car = new Car(new PowerEngine(bigCylinders), new Tires());
```

The point here is “*the Car class did not have to change at all*”.

# Dependency Injection

---

Car class is much easier to test now because car dependencies are in control.

Now Tester can pass mocks to constructor during each test:

```
class MockEngine extends Engine {
 cylinders = 8;
}
class MockTires extends Tires {
 make = 'YokoGoodStone';
}
let car = new Car(new MockEngine(), new MockTires());
```

Dependency Injection is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

# Service

---

1. Scenarios in which same code needs to be used in multiple components.
2. Scenarios where data needs to be shared between components.

Angular service is to handle the above scenario. Use angular CLI to generate a service **ng g service data**

*installing service*

*create src\app\data.service.spec.ts*

*create src\app\data.service.ts*

*WARNING Service is generated but not provided, it must be provided to be used*

The warning simply means that have to add it to the providers property of NgModule decorator in app.module.ts



# Service

---

Open the app.module.ts

```
import { DataService } from './data.service';

@NgModule({
 [DataService],
})
```

Open the service file generated by Angular CLI

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

@Injectable()
export class DataService {
 constructor() { }
}
```

It is similar to component, it imports Injectable as opposed to a Component. Injectable decorator lets Angular know if Angular needs to inject other dependencies into service.

# Service

---

This service connect to a database via http protocol to return results. Let us hardcode our array and create a simple method to access the data in the array.

```
export class DataService {
 constructor() {}
 cars = [
 'Ford','Chevrolet','Buick'
];
 getData():string[] {
 return cars;
 }
}
```

# Service

---

To consume the service let us open the component file

**Step 1:** Import the service at the top of the component

```
import { DataService } from './data.service';
```

**Step 2:** Modify the constructor so that the service can be injected

```
export class AppComponent {
 constructor(private dataService:DataService) {
 }
}
```

Now properties and methods of `dataService` can be accessed in the component.

# Service

---

**Step 3:** To access the data underneath the constructor add the `ngOnInit()` lifecycle hook, which loads the data

```
someData:String[];
ngOnInit() {
 this.someData = this.dataService.getData();
 console.log(this.someData);
}
```

**Step 4:** In the template property of the component decorator

```
@Component({
 template: `<p>{{ someData }}</p>`
})
```



PIPES

# Introduction

---

An application starts out with

1. Get data
2. Transform them
3. Show them to users

Once data arrived the raw data can be pushed directly to the view, but that rarely makes for a good user experience.

For example users prefer to see a date like April 15, 1988 rather than Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time).

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

# Using pipes

---

Inside the interpolation expression, the component's birthday value through the pipe operator ( `|` ) to the Date pipe function on the right. All pipes work this way.

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

@Component({
 selector: 'hero-birthday',
 template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})

export class HeroBirthdayComponent {

 birthday = new Date(1988, 3, 15);

}
```

# Built-in pipes

---

Angular comes with a stock of pipes such as

1. CurrencyPipe
2. DatePipe
3. DecimalPipe
4. JsonPipe
5. LowerCasePipe
6. PercentPipe
7. SlicePipe
8. TitleCasePipe
9. UpperCasePipe



# Parameterizing a pipe

---

A pipe can accept any number of optional parameters to fine tune its output.

1. To add parameters to a pipe, follow the pipe name with a colon ( : ) and then the parameter value (such as currency:'EUR').
2. If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5).

*<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>*

# Parameterizing a pipe

---

A button click event is attached with toggleFormat() method. Method toggles format property between a short and a longer form.

*template: `<p>The hero's birthday is {{ birthday | date:format }}</p>*

*<button (click)="toggleFormat()">Toggle Format</button>`*

*export class HeroBirthday2Component {*

*birthday = new Date(1988, 3, 15);*

*toggle = true;*

*get format() { return this.toggle ? 'shortDate' : 'fullDate'; }*

*toggleFormat() { this.toggle = !this.toggle; }*

*}*

# Chaining Pipes

---

Pipes can be chained together with useful combinations. To display the birthday in uppercase, the birthday is chained to the DatePipe and on to the UpperCasePipe.

```
{{ birthday | date | uppercase }}
```

The birthday displays as **APR 15, 1988**.

Chains the same pipes as above, but passes in a parameter to date as well.

```
{{ birthday | date:'fullDate' | uppercase }}
```

The birthday displays as **FRIDAY, APRIL 15, 1988**

# Custom Pipes

---

Let us create a new custom pipe using the ng command

*ng generate pipe ExponentialStrength*

A generated custom pipe named ExponentialStrengthPipe is already added to the Module

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
 name: 'exponentialStrength'
})
export class ExponentialStrengthPipe implements PipeTransform {
 transform(value: any, args?: any): any {
 return null;
 }
}
```

# Custom Pipes

---

The pipe syntax is `{{ 2 | exponentialStrength: 10 }}`.

Here the 2 is value and 10 is the argument. Let us change the existing transform method as follows.

```
transform (value: number, exponent: string): number {
 let exp = parseFloat(exponent);
 return Math.pow(value, isNaN(exp) ? 1 : exp);
}
```

The pipe class implements the PipeTransform interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.

# Custom Pipes

---

Now let us use the custom pipe in template property of our existing component.

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

@Component({
 selector: 'power-booster',
 template: `<h2>Power Booster</h2>
 <p>Super power boost: {{2 | exponentialStrength: 10}}</p>`
})

export class AppComponent { }
```

Custom pipe is used the same way built-in pipes are used.



# ANIMATIONS



# Transitioning between two states

---

Simple animation that transitions an element between two states can be done in Angular. For animations import animation-specific libraries.

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
imports: [
 BrowserAnimationsModule
],
```

app.component.ts

```
import { Component, Input } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';
```



# Transitioning between two states

---

Let us create an animation trigger called `changeBackground` which does transition and makes an element appears in a lighter colour.

```
animations: [
 trigger('changeBackground', [
 state('inactive', style({ backgroundColor: '#eee' })),
 state('active', style({ backgroundColor: '#cfd8dc' })),
 transition('inactive => active', animate('100ms ease-in')),
 transition('active => inactive', animate('100ms ease-out'))
])
]
```

# Car Model Class

---

In app directory, create Hero model class src/app/car.ts file

```
export class Car {
 constructor (
 public id: number,
 public brand: string,
 public number: string,
 public colour: string,
 public state: string
) {}
 toggleState() {
 this.state = (this.state === 'active' ? 'inactive' : 'active');
 }
}
```

# Transitioning between states

---

Create an array of car objects in the component

```
Cars = [
 new Car(1, 'Proton', 'JCG6939', 'Blue', 'inactive'),
 new Car(2, 'Volvo', 'JCG6940', 'Black', 'inactive'),
 new Car(3, 'Mazda', 'JCG4039', 'White', 'inactive')
]
```

Add animation to template, using the [@triggerName] syntax

```

 <li *ngFor="let car of Cars" [@changeBackground]="car.state"
 (click)="car.toggleState()">{{car.brand}}

```

# Multiple Transitions

---

After defining the states, define *transitions* between the states using the following syntax

```
transition('inactive => active', animate('100ms ease-in')),
transition('active => inactive', animate('100ms ease-out'))
```

If several transitions have same timing configuration, they can be combined using the following syntax

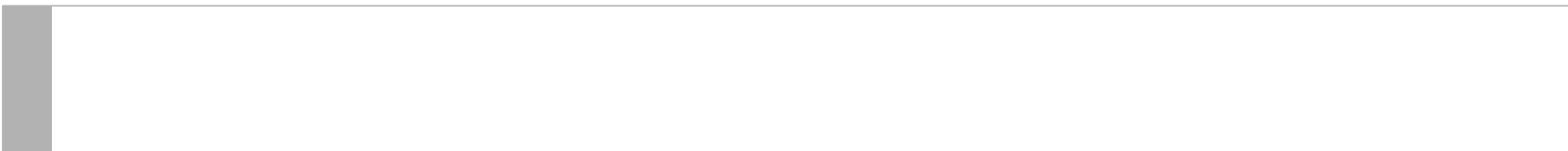
```
transition('inactive => active, active => inactive', animate('100ms ease-out'))
```

In the above case the shorthand syntax `<=>` can be used

```
transition('inactive <=> active', animate('100ms ease-out'))
```



FORMS



# Forms

---

Forms are the mainstay of business applications. Forms to log in, submit a help request, place an order, book a flight, schedule a meeting and perform countless other data-entry tasks. To develop a form, it requires

1. Design skills
2. Coding skills to support two-way data binding, change tracking, validation, and error handling

Let us start building an Angular Form with a component and template and use `ngModel` to create two-way data bindings for reading and writing input-control values.

# Forms

---

Build the web form in small steps:

1. Create the Car model class.
2. Create the component that controls the form.
3. Create a template with the initial form layout.
4. Bind data properties to each form control using the `ngModel` two-way data-binding syntax.
5. Add a name attribute to each form-input control.
6. Add custom CSS to provide visual feedback.
7. Show and hide validation-error messages.
8. Handle form submission with *`ngSubmit`*.
9. Disable the form's *Submit* button until the form is valid.

# Car Model Class

---

In app directory, create Hero model class src/app/car.ts file

```
export class Car {
 constructor (
 public id: number,
 public brand: string,
 public number: string,
 public colour: string
) {}
}
```

As users enter form data, angular captures the changes and update an instance of a model. A Car model class is a property bag that holds Car with its four required properties such as id, name, number and colour.



# Module - Configuration

---

1. To create forms first import FormsModule from @angular/forms.
2. Add FormsModule to the “imports” list defined in @NgModule decorator.

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

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

```
imports: [
```

```
 BrowserModule,
```

```
 FormsModule
```

```
],
```

# Component Class

---

1. In the component import the Data Model

```
import { Car } from './car';
```

2. Create the colours array

```
brands = ['proton', 'mazda', 'volvo'];
```

```
colours = ['white', 'black', 'blue'];
```

3. Create the myCar object

```
myCar = new Car(42, this.brands[0], 'JCG6939', this.colours[0]);
```

4. Add a method diagnostic returns a JSON representation of Model

```
get diagnostic() { return JSON.stringify(this.myCar); }
```

# Create an initial HTML Form

---

Create template file

```
<div class="container">
```

```
<h1>Car Form</h1>
```

```
<form>
```

```
<div class="form-group">
```

```
 <label for="name">ID</label><input type="text" class="form-control" id="id" required>
```

```
</div>
```

```
<div class="form-group">
```

```
 <label for="number">Number</label><input type="text" class="form-control" id="number">
```

```
</div>
```

```
<button type="submit" class="btn btn-success">Submit</button>
```

```
</form>
```

```
</div>
```



# Two way data binding

---

myCar data not appearing in the form, because it is not bind to form. Now change both textbox as follows

```
<input type="text" class="form-control" id="id" [(ngModel)]="myCar.id"
name="id" required>{{ myCar.id }}
```

```
<input type="text" class="form-control" id="number"
[(ngModel)]="myCar.number" name="number" required>{{ myCar.number }}
```

To display data declare a template variable for <form> tag.

```
<form #carForm="ngForm">
```

Now the variable carForm is reference variable to NgForm directive that governs the form as a whole.

# Two way data binding

---

Do the two way data binding on select element also.

```
<select class="form-control" id="brand" [(ngModel)]="myCar.brand"
name="brand" required>{{ myCar.brand }}
```

```
<select class="form-control" id="colour" [(ngModel)]="myCar.colour"
name="colour" required>{{ myCar.colour }}
```

Let us remove all the interpolation and display the input box binding message at the top of the component using the diagnostic property.

```
{{ diagnostic }}
```

The code confirms two-way data binding works for entire car model.

# Track control state and validity

---

1. The ngModel directive in form allows to identify whether user has touched the control or value has changed or value became invalid.
2. The ngModel directive updates the control with special Angular CSS classes that reflect the state. Those class name can be used to change the appearance of the control.

State	Class if true	Class if false
The control has been visited.	<i>ng-touched</i>	<i>ng-untouched</i>
The control's value has changed.	<i>ng-dirty</i>	<i>ng-pristine</i>
The control's value is valid.	<i>ng-valid</i>	<i>ng-invalid</i>

# Track control state and validity

---

Add a template reference variable named `spy` to the `<input>` tag and add the `spy.className` interpolation to display the CSS classes.

```
<input type="text" class="form-control" id="id" [(ngModel)]="myCar.id"
name="id" #spy required>{{ myCar.id }}{{ spy.className }}
```

Now run the app and at the input box do exactly as follows

1. Look but don't touch.
2. Click inside the name box, then click outside it.
3. Add slashes to the end of the name.
4. Erase the name.



# Add CSS for visual feedback

---

Using track feature, required fields and invalid data can be visually highlighted. Effects can be achieved by adding CSS class.

```
.ng-valid[required], .ng-valid.required {
 border-left: 5px solid #42A948;
}
```

```
.ng-invalid:not(form) {
 border-left: 5px solid #a94442;
}
```

# Show / hide validation error messages

---

Visually highlighted but no message appears. When it is empty, user must know what to do.

```
<label for="id">ID</label>
```

```
<input type="text" class="form-control" id="id" [(ngModel)]="myCar.id"
 name="id" #id=ngModel required>
```

```
<div [hidden]="id.valid || id.pristine" class="alert alert-danger">
 ID is required</div>
```

To access the input box within the template, a reference variable is required. Let us remove the #spy and add #id=ngModel. Here the variable is called id and the value "ngModel"

# Manage car in the Form

---

Place a *New Car* button at the bottom of form and bind its click event to a *newCar* component method. In template file

```
<button type="button" class="btn btn-default" (click)="newCar()">
```

```
New Car</button><div>{{ post_value }}</div>
```

In component file

```
myNewCar;
```

```
newCar() { this.myNewCar = JSON.parse(JSON.stringify(this.myCar)); }
```

```
get post_value() { return JSON.stringify(this.myNewCar); }
```

# Disable the Submit button

---

Users are able to submit form after filling it in. The Submit button will trigger a form submit. A form can be submitted only upon overall validity of all fields. All fields can be accessed via carForm variable.

```
<button type="submit" class="btn btn-success" (click)="newCar()"
 [disabled]="!carForm.form.valid">New Car</button>
```

Component's newCar method can be bind with form's ngSubmit event. The click event at the button can be removed.

```
<form (ngSubmit)="newCar()" #carForm="ngForm">
```



# Reactive Forms

---

To use reactive forms, we need to **import the ReactiveFormsModule** into our parent module.

```
import { BrowserModule } from '@angular/platform-browser'
import { NgModule } from '@angular/core'
import { ReactiveFormsModule } from '@angular/forms'
import { AppComponent } from './app.component'
@NgModule({
 declarations: [AppComponent],
 imports: [BrowserModule, ReactiveFormsModule],
 providers: [],
 bootstrap: [AppComponent],
})
export class AppModule {}
```

# Reactive Forms

---

Let us create a simple contact-form. To do so, first create a model.

```
export class ContactRequest {
 personalData: PersonalData
 requestType: any = ""
 text: string = ""
}
```

```
export class PersonalData {
 email: string = ""
 mobile: string = ""
 country: string = ""
}
```

# Reactive Forms

---

1. Binding the model directly to the form allows altering the original data directly.
2. With reactive forms we keep the data in the form model, until the user hits the submit button.
3. Upon hitting the submit button, the data is copied over to the original model, replacing everything.
4. This type of overwriting is called "**immutable objects**".
5. To create this form-model, angular uses a class called **FormGroup**
6. The constructor of this FormGroup class then takes an object, that can contain sub-form-groups and **FormControls**.



# Reactive Forms

---

Let us create a new method called `createFormGroup`, that creates a new instance of the form-model.

```
createFormGroup() {
 return new FormGroup({
 personalData: new FormGroup({
 email: new FormControl(),
 mobile: new FormControl(),
 country: new FormControl()
 }),
 requestType: new FormControl(),
 text: new FormControl()
 });
}
```

# Reactive Forms

---

1. The country and the request Type fields to be select-able as a drop-down.

Let us provide some options to choose from:

```
countries = ['USA', 'Germany', 'Italy', 'France']
```

```
requestTypes = ['Claim', 'Feedback', 'Help Request']
```

2. Let us save the result of the createFormGroup to a public field, to be accessible in our template.

```
contactForm: FormGroup;
```

```
constructor() {
```

```
 this.contactForm = this.createFormGroup();
```

```
}
```

# Reactive Forms

---

**Implementing the Template:** The HTML look just like a regular form. However, we have to tell angular, which FormGroup and which FormControl to use for each control.

```
<form [formGroup]="contactForm" (ngSubmit)="onSubmit()" novalidate>
 <div formGroupName="personalData" novalidate>
 <input formControlName="email" />
 <input formControlName="mobile" />
 <select formControlName="country">
 <option *ngFor="let country of countries" [value]="country">
 {{country}}</option>
 </select>
 </div>
```

# Reactive Forms

---

```
<select formControlName="requestType">
 <option *ngFor="let requestType of requestTypes"
 [value]="requestType">
 {{requestType}}
 </option>
</select>
<input formControlName="text" />
<button type="submit" [disabled]="contactForm.pristine">
 Save</button>
<button type="reset" (click)="revert()"
 [disabled]="contactForm.pristine">Revert</button>
</form>
```

Let us run the app using **ng serve**

# Reactive Forms

---

Extracting the Data from the Form

```
onSubmit() {
 // Make sure to create a deep copy of the form-model

 const result: ContactRequest =
 Object.assign({}, this.contactForm.value);

 result.personalData = Object.assign({}, result.personalData);

 // Do useful stuff with the gathered data

 console.log(result);
}
```

# Reactive Forms

---

1. To reset the form, we can either use the form's reset method without any parameter.

// Resets to blank object

```
this.contactForm.reset();
```

2. Call the reset method by passing a set of default parameters along:

// Resets to provided model

```
this.contactForm.reset({ personalData: new PersonalData(),
 requestType: "", text: "" });
```



**HTTP CLIENT**

# Introduction

---

Front-end applications communicate with backend services over HTTP protocol. Browsers support two APIs for making HTTP requests

1. **XMLHttpRequest interface**
2. **fetch() API**
  - HttpClient module extend XMLHttpRequest interface exposed by browsers
  - HTTPClient provides API to develop application that require HTTP functionalities
  - HttpClient includes testability support, strong typing of request and response objects, request and response interceptor support, and better error handling



# Setting up HttpClient

---

Before using the HttpClient, must install the HttpClientModule which provides it.

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {HttpClientModule} from '@angular/common/http';
@NgModule({
 imports: [
 BrowserModule,
 HttpClientModule,
],
})
export class AppModule {}
```

# Request via HttpClient

---

Once HttpClientModule is imported into app module, HttpClient can be injected into components and services.

```
import { HttpClient } from '@angular/common/http';
import { OnInit } from '@angular/core';
export class MyComponent implements OnInit {
 constructor(private http: HttpClient) {}
 ngOnInit(): void {
 this.http.get('https://www.reddit.com/r/funny/.json').subscribe(data => {
 this.results = data['data'].children;
 });
 }
}
<div *ngFor="let result of results">{{ result.data.title }}</div>
```

# Error in Response

---

Detecting what error actually occurred is more important.

```
this.http.get ('https://www.reddit.com/r/funnies/.json').subscribe (
 data => {
 this.errorMessage = '';
 this.results = data['data'].children;
 },
 err => {
 if (err.error instanceof Error) {
 this.errorMessage = 'Error occurred : ' + err.error.message;
 } else {
 this.errorMessage = 'Backend returned code ' +
 err.status + ', body was: ' + err.error.message;
 }
 });
```

# Non-JSON data

---

Not all APIs return JSON data. For instance, the [api.ipify.org](https://api.ipify.org) website returns the IP address in text format. Whenever a server returns the response as a text or text file, the code has to tell HttpClient that expected result is a textual response

```
this.http.get ('https://api.ipify.org', {responseType: 'text'}).subscribe (
 data => {

 this.results = data;
 });
```

Add the variable to the template

```
<div>{{ results }}</div>
```

# Post data to Backend

---

Submitting a form triggers data to be POST to a server. The code for sending a POST request is very similar to the code for GET.

```
welcomeMessage = "";
constructor (private http:HttpClient) {
};

postData(firstname):void {
 this.http.post('http://localhost:7777/test.php', firstname,
 {responseType: 'text'}).subscribe(
 data => {
 this.welcomeMessage = data;
 }
);
}
```

# Post data to Backend

---

Code in the Template

```
<div>First Name : </div>
```

```
<div>
```

```
<input type="text" #firstname (keyup.enter)="postData(firstname.value)"/>
```

```
</div>
```

```
<div>{{ welcomeMessage }}</div>
```

Create a PHP Code and run the built-in PHP server

*Php Output - Data from Angular*

```
[<?php
```

```
 $json_str = file_get_contents('php://input');
```

```
 echo ($json_str);
```

```
?>]
```



The diagram consists of two horizontal rectangular bars. The top bar is light gray and contains the word "ROUTER" in black, bold, uppercase letters on its right side. The bottom bar is a darker gray and is empty. Both bars have a thin black border.

**ROUTER**

# Overview

---

The browser is a familiar model of application navigation:

1. Enter a URL in the address bar and the browser navigates to a corresponding page based on URL typed.
2. Click links on the page and the browser navigates to a new page.
3. Click browser's back and forward buttons and browser navigates backward and forward through the history of pages you've seen.

The Angular Router borrows from this model and interprets a URL as instruction to navigate to a client-generated view. It pass parameters along to supporting view component.



# Router Setup

---

## *<base href>*

Applications should add a `<base>` element to `index.html` as first child in `<head>` tag to tell the router how to compose navigation URLs.

*<base href="/">*

## **Router imports**

In Angular, Router is an optional service that presents a particular component view for a given URL.

*import { RouterModule, Routes } from '@angular/router';*

# Router Setup

---

Create 2 components

1. *ng generate component CarManager*
2. *ng generate component DriverManager*

Both components are defined with paths in a routes array.

```
const appRoutes:Routes = [
 { path: 'car', component: CarManagerComponent },
 { path: 'driver', component: DriverManagerComponent }
];
```

Pass appRoutes array to RouterModule.forRoot method

```
imports: [
 RouterModule.forRoot(appRoutes)
]
```

# Router Outlet / Links

---

## Router outlet

When browser URL becomes /car, router matches that URL to route path /car and displays CarManagerComponent after a RouterOutlet

## Router links

URL arrive as a result of user action such as click of an anchor tag.

```
<nav>
```

```
 Car
```

```
 Driver
```

```
</nav>
```

```
<router-outlet></router-outlet>
```

# Wildcard Router

---

Any URL other than those declared in module causes router to throw an error and crash the app. A wildcard route can be added to intercept invalid URLs and handle them gracefully.

```
const appRoutes:Routes = [
 { path: 'car', component: CarManagerComponent },
 { path: 'driver', component: DriverManagerComponent },
 { path: '**', component: PageNotFoundComponent }
];
```

Create a PageNotFoundComponent to handle any invalid URLs

# Default Route

---

1. When the application launches, the initial URL in the browser bar is something like this localhost:3000
2. This URL does not match any routes which means the router displays the `PageNotFoundComponent`.
3. The application needs a **default route** to a valid page.

```
const appRoutes: Routes = [
 { path: 'car', component: CarManagerComponent },
 { path: 'driver', component: DriverManagerComponent },
 { path: '', redirectTo: '/car', pathMatch: 'full' },
 { path: '**', component: PageNotFoundComponent }
];
```

# Routing Module

---

As application grows, the routing configuration have to refactor into its own file. Let us move the Routing information into a module.

*ng g module app --routing*

1. Separates routing concerns from other application concerns.
2. Provides a module to replace or remove when testing the application.
3. Provides a well-known location for routing service providers including guards and resolvers.
4. Does not declare in components.

# Routing Module

---

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { CrisisListComponent } from './crisis-list.component';
import { HeroListComponent } from './hero-list.component';
import { PageNotFoundComponent } from './not-found.component';
const appRoutes: Routes = [
 { path: 'crisis-center', component: CrisisListComponent },
 { path: 'heroes', component: HeroListComponent },
 { path: '', redirectTo: '/heroes', pathMatch: 'full' },
 { path: '**', component: PageNotFoundComponent }
];
@NgModule({
 imports: [RouterModule.forChild(appRoutes)],
 exports: [RouterModule]
})
export class AppRoutingModule {}
```

# Changes to Main Module

---

```
import { AppRoutingModule } from './app-routing.module';
@NgModule({
 imports: [
 BrowserModule,
 FormsModule,
 AppRoutingModule
],
 ...
})
export class AppModule { }
```

Note: Only call `RouterModule.forRoot` in the root `AppRoutingModule` (top level application routes). In any other module, you must call the `RouterModule.forChild` method to register additional routes.



# Route Parameters

---

1. Route parameters are used to specify a parameter value within route URL. URL looks as follows  
*localhost:4200/vehicle*  
*localhost:4200/vehicledetail/15*
2. Actual route looks as follows  
*{ path: 'vehicledetail/:id', component: VehicleDetailComponent }*
3. 15 is the value of the id parameter, which will be used by the corresponding VehicleDetailComponent and present the vehicle whose id is 15.

# Feature Modules

---

1. As your app grows, you can organize code relevant for a specific feature.
2. This helps apply clear boundaries for features.
3. With feature modules, you can keep code related to a specific functionality or feature separate from other code.
4. Delineating areas of your app helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

Make a feature module

**ng generate module system**

# Feature Modules

---

To generate components inside the system module.

**ng generate component system/Header**

Importing the feature modules

1. Open the app.module.ts
2. **import { SystemModule } from './system/system.module';**
3. Add the SystemModule inside the imports array.

# Feature Modules

---

To render HeaderComponent which is under the system module in AppComponent first add the HeaderComponent into the @exports array of system.module.ts as follows

```
import { HeaderComponent } from './header/header.component';
```

```
@NgModule({
```

```
 exports: [
```

```
 HeaderComponent
```

```
]
```

```
})
```

```
export class SystemModule { }
```

# Lazy Modules

---

By default, NgModules are eagerly loaded. As soon as the app loads, all the NgModules are loaded, whether they are immediately necessary or not.

Lazy Loading is for large apps with lots of routes. A design pattern that loads NgModules as needed. Lazy loading helps keep initial bundle sizes smaller, which in turn helps decrease load times.

There are three main steps to setting up a lazy loaded feature module:

1. Create the feature module.
2. Create the feature module's routing module.
3. Configure the routes.

# Lazy Modules

---

1. Setup an app  
ng new customer-app --routing
2. Create a feature module with routing  
ng generate module customers --routing
3. Add a component to the feature module  
ng generate component customers/customer-list
4. Add another feature module  
ng generate module orders --routing
5. Add a component to the feature module  
ng generate component orders/order-list

# Lazy Modules

---

## Setup the UI

```
<h1>{{title}}</h1>
```

```
<button routerLink="/customers">Customers</button>
```

```
<button routerLink="/orders">Orders</button>
```

```
<button routerLink="">Home</button>
```

```
<router-outlet></router-outlet>
```

To see the app in the browser so far, enter the following command in the terminal window:

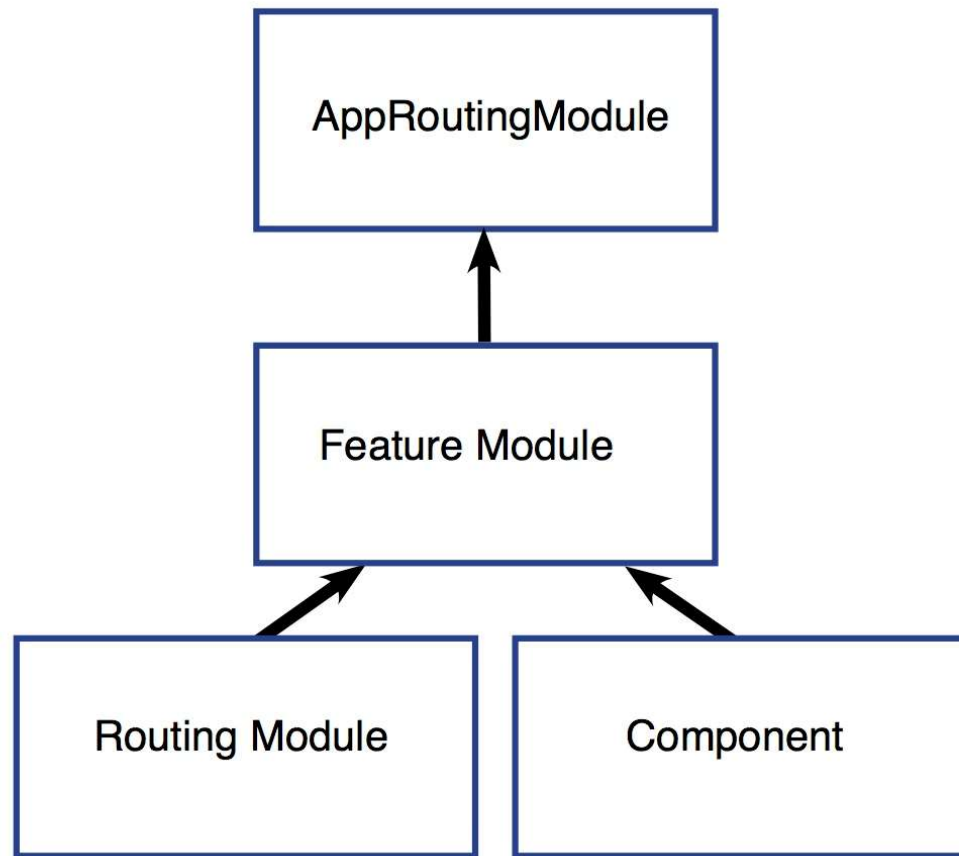
**ng serve**

Then go to localhost:4200 to see “app works!” and three buttons.

# Lazy Modules

---

Configure the routes





# Lazy Modules

---

In AppRoutingModuleModule update the routes array with the following

```
const routes: Routes = [
 {
 path: 'customers',
 loadChildren: './customers/customers.module#CustomersModule'
 }, {
 path: 'orders',
 loadChildren: './orders/orders.module#OrdersModule'
 }, {
 path: "",
 redirectTo: "",
 pathMatch: 'full'
 }
];
```

# Lazy Module

---

## Inside the future module

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CustomersRoutingModule } from './customers-routing.module';
import { CustomerListComponent } from './customer-list/customer-list.component';
@NgModule({
 imports: [
 CommonModule,
 CustomersRoutingModule
],
 declarations: [CustomerListComponent]
})
export class CustomersModule { }
```

# Lazy Module

---

Configure the feature module's routes

```
const routes: Routes = [{
 path: "",
 component: CustomerListComponent
}];

@NgModule({
 imports: [RouterModule.forChild(routes)],
 exports: [RouterModule]
})

export class CustomersRoutingModule { }
```

# Lazy Module

---

Configuring the Routes array for the orders-routing.module.ts

```
import { OrderListComponent } from './order-list/order-list.component';

const routes: Routes = [{
 path: "",
 component: OrderListComponent
}];
```



# SIMPLE PROJECT



# Vehicle Management System

---

1. Create a new project using the following command  
*ng new VehicleManagement*
2. Create three components using the following command  
*ng generate component About*  
*ng generate component Car*  
*ng generate component CarDetail*  
*ng generate component Lorry*  
*ng generate component LorryDetail*
3. Create two services service using the command  
*ng generate service Car*  
*ng generate service Lorry*

# Car Service

---

```
import { Injectable } from '@angular/core';
export class Car {
 constructor(public id: number, public number: string,
 public brand: string, public colour: string) {
 }
}

let Cars = [
 new Car(11, 'JCG6939', 'Proton Saga', 'Dark Blue'),
 new Car(12, 'SAB3708', 'Proton Wira', 'White'),
 new Car(13, 'WKL7373', 'Mazda', 'Light Red')
];

@Injectable()
export class CarService {
 constructor() { }
}
```

# Car Service

---

Add two methods to CarService

```
export class CarService {
 constructor() { }
 getCars() {
 return Cars;
 }
 getCar(id:number | string) {
 return Cars.find(function (car) { return car.id === id; });
 }
}
```

Declare the CarService inside the providers list of the AppModule.

```
import { CarService } from './car.service';
providers: [CarService],
```



# Lorry Service

---

```
import { Injectable } from '@angular/core';
export class Lorry {
 constructor(public id: number, public number: string,
 public brand: string, public colour: string) {
 }
}

let Lorries = [
 new Lorry(21, 'PGG6939', 'Nissan', 'Yellow'),
 new Lorry(22, 'SKK3708', 'Volvo', 'Red'),
 new Lorry(23, 'WKK7373', 'Mazda', 'Red')
];

@Injectable()
export class LorryService {
 constructor() { }
}
```

# Lorry Service

---

Add two methods to LorryService

```
export class LorryService {
 constructor() { }
 getLorries() {
 return Lorries;
 }
 getLorry(id:number | string) {
 return Lorries.find(function (lorry) { return lorry.id === id; });
 }
}
```

Declare the LorryService inside the providers list of the AppModule.

```
import { LorryService } from './lorry.service';
providers: [LorryService],
```

# Create Routes

---

1. The Landing page will have 3 tabs. Each tab is linked to a component and the URL looks as follows

*localhost:4200/about*

*localhost:4200/car*

*localhost:4200/lorry*

2. Create the AppRoutingModuleModule using the following command

*ng g module app –routing*

Move app-routing.modules.ts file to src/app folder.

3. Import the AppRoutingModuleModule into AppModule

*import { AppRoutingModuleModule } from './app-routing.module';*

*imports [ AppRoutingModuleModule ],*

# Create Routes

---

4. Add the required routes into AppRoutingModuleModule

```
import { AboutComponent } from './about/about.component';
import { CarComponent } from './car/car.component';
import { LorryComponent } from './lorry/lorry.component';
const routes: Routes = [
 { path: 'about', component: AboutComponent },
 { path: 'car', component: CarComponent },
 { path: 'lorry', component: LorryComponent }
];
@NgModule({
 imports: [RouterModule.forRoot(routes)],
 exports: [RouterModule]
})
```

5. Since this is main routing module we use *RouterModule.forRoot*

# Create Router Links

---

Modify the app component template as follows

```
<div style="text-align:center">
```

```
<h1 class="title">Vehicle Management System</h1>
```

```
</div>
```

```
<nav>
```

```
About
```

```
Car
```

```
Lorry
```

```
</nav>
```

```
<router-outlet></router-outlet>
```

# Merge detail component

---

1. Move the following files from car-detail folder to car folder

*car-detail.component.css*

*car-detail.component.html*

*car-detail.component.ts*

2. Update the AppModule from

*import { CarDetailComponent } from './car-detail/car-detail.component';*

To

*import { CarDetailComponent } from './car/car-detail.component';*

3. Delete the folder car-detail

# Merge detail component

---

1. Move the following files from lorry-detail folder to car folder

*lorry-detail.component.css*

*lorry-detail.component.html*

*lorry-detail.component.ts*

2. Update the AppModule from

*import { LorryDetailComponent } from './lorry-detail/lorry-detail.component';*

To

*import { LorryDetailComponent } from './lorry/lorry-detail.component';*

3. Delete the folder lorry-detail

# Create Route Modules

---

1. Create a new routing module file for the Car Module  
*ng g module car -routing*
2. Add the required routing in the car-routing.module.ts  
*import { CarDetailComponent } from './car-detail.component';*  
*const routes: Routes = [*  
*{ path: 'car-detail/:id', component: CarDetailComponent }*  
*];*
3. Do the same for Lorry Module  
*ng g module lorry -routing*  
*import { LorryDetailComponent } from './lorry-detail.component';*  
*const routes: Routes = [*  
*{ path: 'lorry-detail/:id', component: LorryDetailComponent }*  
*];*
4. The car.module.ts and lorry.module.ts has references to routing module. Import these two modules inside AppModule.



# Car Component

---

1. Import the Car services and get all cars

```
import { Car, CarService } from '../car.service';
cars;
constructor(private carService:CarService) {
}
ngOnInit() {
 this.cars = this.carService.getCars();
}
```

2. In car template code as follows

```
<h3>Cars</h3>
<div *ngFor="let car of cars">
 {{ car.brand }}
</div>
```

# Lorry Component

---

1. Import the Lorry services and get all lorries

```
import { Lorry, LorryService } from '../lorry.service';
lorries;
constructor(private lorryService:LorryService) {
}
ngOnInit() {
 this.lorries = this.lorryService.getLorries();
}
```

2. In lorry template code as follows

```
<h3>Lorries</h3>
<div *ngFor="let lorry of lorries">
 {{ lorry.brand }}
</div>
```

# Creating Route Parameters

---

1. Create a new event in car component template as follows

```
<h3>Cars</h3>
```

```
<div *ngFor="let car of cars" (click)="onSelect(car)">
```

```
{{ car.id }} {{ car.number }} - {{ car.brand }}
```

```
</div>
```

2. Similarly create a new event in lorry component template

```
<h3>Lorries</h3>
```

```
<div *ngFor="let lorry of lorries" (click)="onSelect(lorry)">
```

```
{{ lorry.id }}
```

```
{{ lorry.number }} - {{ lorry.brand }}
```

```
</div>
```

# Creating Route Parameters

---

3. Add the onSelect method in car component

```
import { Router } from '@angular/router';

constructor(private carService:CarService, private router:Router) { }

onSelect(car: Car) {

 this.router.navigate(['/car-detail', car.id]);

}
```

4. Add the onSelect method in lorry component

```
import { Router } from '@angular/router';

constructor(private lorryService:LorryService, private router:Router) { }

onSelect(lorry: Lorry) {

 this.router.navigate(['/lorry-detail', lorry.id]);

}
```

# Car Detail Component

---

1. Fix the CarDetailComponent

```
import { Car, CarService } from '../car.service';
export class CarDetailComponent implements OnInit {
 car;
 constructor(private carService:CarService) { }
 ngOnInit() { this.car = this.carService.getCar(11); }
}
```

2. Fix the CarDetailComponent template

```
<h2>CAR</h2>
<div *ngIf="car">
 <div><label>Id: </label>{{ car.id }}</div>
 <div><label>Brand: </label>{{ car.brand }}</div>
 <div><label>Number: </label>{{ car.brand }}</div>
 <div><label>Colour: </label>{{ car.colour }}</div>
</div>
```

# Lorry Detail Component

---

1. Fix the LorryDetailComponent

```
import { Lorry, LorryService } from '../lorry.service';
export class LorryDetailComponent implements OnInit {
 lorry;
 constructor(private lorryService:LorryService) { }
 ngOnInit() {
 this.lorry = this.lorryService.getLorry(11);
 }
}
```

2. Fix the LorryDetailComponent template

```
<h2>LORRY</h2>
<div *ngIf="lorry">
 <div><label>Id: </label>{{ lorry.id }}</div>
 <div><label>Brand: </label>{{ lorry.brand }}</div>
 <div><label>Number: </label>{{ lorry.brand }}</div>
 <div><label>Colour: </label>{{ lorry.colour }}</div>
</div>
```

# Accepting Route Parameters

---

The `ActivatedRoute` is a service has a `params` property which is an array of parameter values, indexed by name. To use `ActivatedRoute`, we need to import it in our component.

```
import { ActivatedRoute } from '@angular/router';
```

Then inject it into the component using dependency injection

```
constructor(private _ActivatedRoute:ActivatedRoute)
```

```
ngOnInit() {
```

```
 this.id=this._ActivatedRoute.snapshot.params['id'];
```

```
 this.car = this.carService.getCar(this.id);
```

```
}
```



# THANK YOU



ANGULAR 13