**How We Move with the course :**

**A) What and Why Angular :**

**Angular 1 Came in 2010 . Angular 2 - 2016 Angular 4 2016 Dec end. Ang 5 2017 Ang 6 May 2018.**

**Ang7 Nov 2018**

**Angular1 is called as angularjs and Angualar2+ is called Angular only**

* Today, Angular is thriving, following Angular 2’s launch in September 2016.
* It is created and maintained by Google team.
* Angular is well-suited for large and complex applications.
* Ember and ReactJS are two competitors to Angular’s frameworks.
* Ember is fully featured. Ember does not provide developers with a way to run code on multiple platforms; as Ember is a framework for Web apps only.
* Yes, Angular is not the most simple or easy-to-use framework out there. It takes time to fully understand some of the concepts that Angular builds on.
* With Angular, developers have an amazingly robust, well-documented framework to build powerful applications.
* Angular presents you not only the tools but also design patterns to build your project in a maintainable way.
* It’s a good idea to get the grasp of the Angular CLI to speed up the development process even further.
* Node’s package manager npm is used extensively to install Angular itself and other components, so you’ll need to be comfortable with that as well.
* While developing the app, it’s vital to be able to debug the code, so you should know how to work with debugging tools like Augury.
* Angular helps build interactive and dynamic single page applications (SPAs) with its compelling features including templating, two-way binding, modularization, RESTful API handling, dependency injection, and AJAX handling. Designers can use HTML as template language and even extend HTML’ syntax to easily convey the components of the application.

Angular ensures easy development as it eliminates the need for unnecessary code. It has a simplified MVC architecture, which makes writing getters and setters needless. Directives can be managed by some other team, as these are not part of app code. All in all, developers are promised less coding, along with lighter and faster apps. And according to Amazon, every 100-millisecond improvement in page loading speed led to 1% increase in revenue.

**Modular Structure**

Angular organizes code into buckets, whether it is components, directives, pipes, or services. Those who are familiar with Angular refer to these buckets as modules. Modules make application functionality organization easy, segregating it into features and reusable chunks. Modules also allow for lazy loading, which paves way for application feature loading in the background or on-demand.

Angular makes it an achievable goal to divide the labor across different team members while ensuring organized code. You can make the best of modules when you have a proper understanding of these. Developers can improve productivity with appropriate modules built.

**Code Consistency**

Any code base requires consistent coding. A writer knows how important consistency is in their content pieces. We know if the content fails to resonate with the readers to a deeper level at any touchpoint, we are on a downward slope of lead conversions. Coding is no different.

Inconsistent coding increases the risks of delayed launches or elevated costs. Unlike it, consistent coding has several benefits, such as it makes sites easier to use and enables the use of templates or pre-defined code snippets.

Angular framework is based on components, which begin in the same style. For instance, each component places the code in a component class or defines a @Component decorator (metadata). These components are small interface elements independent of each other, and thus, offer you several benefits, including:

**Reusability**

The component-based structure of Angular makes the components highly reusable across the app. You can build the UI (User Interface) with moving parts, while also ensuring a smooth development process for developers.

**Simplified Unit-Testing**

Being independent of each other, the components make unit testing much easier.

**Improved Readability**

Consistency in coding makes reading the code a piece of cake for new developers on an ongoing project, which adds to their productivity.

**Ease of Maintenance**

Decoupled components are replaceable with better implementations. Simply put, it enables efficient code maintenance and update.

**NOTE :** Angular is being developed in a MonoRepo it means a single repo for everything. @angular/core, @angular/compiler, @angular/router etc are in the same repo and may have their own versions.

**The advantage of MonoRepo is, you don’t have to deal with the versioning of the code dependencies.**

**Why Ang 3 not released :**

**Now the problem is with the @angular/router which is already in a 3.X version. And that’s because of some active and huge developments on the router section, like route-preload.**

**Now releasing Angular as version 3, with it’s route on version 4 will create confusion**

**To avoid this confusion they decided to skip the version 3 and release with version 4.0.0, so that every major dependency in the MonoRepo are on the right track.**

**Due to this misalignment of the router package’s version, the team decided to go straight for Angular v4.**

**Now the problem is with the @angular/router which is already in a 3.X version. And that’s because of some active and huge developments on the router section, like route-preload.**

**Now releasing Angular as version 3, with it’s route on version 4 will create confusion (this confusion is already there, that’s why they want to resolve it). To avoid this confusion they decided to skip the version 3 and release with version 4.0.0, so that every major dependency in the MonoRepo are on the right track.**

Then Angular5 was released but it was just updated version of Angular4. It has a bunch of additional features and bug fixes like Build Optimizer, HttpClient etc.

Angular6 is the latest version of angular which was released in 2018. It focuses more on toolchain like with Angular in the future, like: ngAngular Elements, CLI Workspaces, Library Support, Angular Material Starter Components. So these are versions of Angular till date.Angular6 is much faster to its previous versions as it is using Ivy Rendered which is a rendering engine.

**ANGULAR 5**

Angular 5 showed up on November 1, 2017. With the promise of speed and a smaller size, it gave us the following features.

**The HttpClient**

One of the greatest anticipated changes from version 4.3 was being able to say goodbye to the Http library.

Instead, they introduced the HttpClient API, which is faster, more secure and efficient than its predecessor. Although this API came with the 4.3 version update, as of Angular 5, Http library was depreciated.

***Some Great Benefits of the HTTPCLIENT API***

* Response body access included support for JSON types and was typed synchronous.
* JSON became an assumed default and no longer had to be parsed explicitly.
* With the use of Interceptors, you could use middleware logic and insert it in the pipeline.
* Request/response objects were immutable.
* Request upload and response download could utilize progress events.

**Angular 6**

Angular 6 was released on May 4th, 2018. That is 6 months after its predecessor's (Angular 5) release. The highlights of Angular 6 include the Angular Command Line Interface (CLI), The Component Development KIT (CDK) and the Angular Material package update. The cherry on top, all three are shipped as part of Angular 6, not separate updates.

Let's get to the gritty details by first knowing that Angular 6 uses the RXJS library, so hurray for reactive programming for web!

Moving on, this version release is more focused on the tooling and support rather than the whole framework.

**Angular Elements**

Remember the Elements package? Angular 6 fully supports it now. What it did was allow us to use Angular components outside of Angular like in JQuery or VueJS apps.

This package primarily focuses on taking an advantage of web components that are supported by all modern web browsers (except Edge). Using the Elements Package, you can create Angular components and publish them as Web Components, which can then be used in any HTML page.

Turning a component into a custom element gives you an easy path for creating dynamic HTML content for your Angular app, and, using the Angular Elements package, it is even easier to create native custom elements.

**Component Dev Kit (CDK)**

The CDK was released in December of 2017, but the Angular Team has made some really neat improvements to it for the 6th version.

With the CDK you can now build your own library of UI components without using the Angular Material library. It also supports Responsive Web Design layouts so you don't have to use other libraries like Flex Layout or even learn using the CSS Grid. It covers them all.

Another brilliant improvement in the CDK includes the @angular/cdk/overlay package. This one has a new positioning logic that makes your pop-ups stay on screen very brilliantly.

**Architecture Overview of Angular**



**B)**

**User Experience similar to a Desktop Application**

* **Productivity and Tooling**
* **Performance**
* **Community**
* **Full-featured Framework**
* **Platform for Targeting Native Mobile not just Web Browsers**

C)

**AngularJS (Angular 1.x)**

**Angular**

**Angular 2**

**Angular 4**

**Angular 5**

**Angular 6**

**C) Angular 2**

Angular 2 was released at the end of 2015. Let's take a look at why this version was released and what it added to web development.

This version of Angular was more focused on the development of mobile apps, as it allowed developers to create cross platform applications. The reason is that it is easier to handle the desktop component of things after the challenges connected to mobile apps (functionality, load time, etc.) have been addressed.

Numerous modules were eliminated out of Angular's core, which led to better performance. These made their way to Angular's ever-growing ecosystem of modules, which means that you have the ability to select and choose the components you want.

Angular 2.0 was aimed at ES6 and "evergreen" modern browsers (these automatically update to the most recent version). Building for these browsers means various hacks and workarounds that make Angular harder to develop can be eliminated, allowing developers to concentrate on the code linked to their company domain.

**Angular 2 Features and Performance**

AtScript is a superset of ES6 and it was used to help develop Angular 2. It is processed from the Traceur compiler (combined with ES6) to generate ES5 code and utilizes TypeScript's syntax to create runtime type assertions rather than compile time tests. But, AtScript is not mandatory--you still have the ability to use plain JavaScript/ES5 code rather than AtScript to compose Angular apps.

**Improved Dependency Injection (DI):**

Dependency injection (a program design pattern where an item is passed its own dependencies, as opposed to producing them) was among the aspects that originally differentiated Angular from its competitors. Dependency Injection is very helpful when it comes to modular development and element isolation, yet its implementation has been plagued with issues since Angular 1.x. Angular 2 handled these problems, in addition to adding missing features like kid injectors along with lifetime/scope control.

**Annotation:**

AtScript supplies tools for linking metadata with functions. This eases the building of object instances by supplying the essential information into the DI library (that will check for related meta data if calling a function or creating the instance of a class). It'll also be simple to override parameter information by providing an Inject annotation.

**Child Injectors:**

A kid injector inherits all of the professional services of its parent together with the capacity to override them at the child level. According to demand, several kinds of objects could be called out and mechanically overridden in a variety of scopes.

**Instance Scope:**

The enhanced DI library is comprised of instance scope controllers, which are even stronger when used with child injectors along with your scope identifiers.

**Dynamic Loading:**

This is a feature which was not available in the previous version(s) of Angular. It was addressed by Angular 2, however, which allowed programmers to add new directives or controls on the fly.

**Templating:**

In Angular 2, the template compilation procedure is asynchronous. Since the code relies on the ES6 module, the module loader will load dependencies simply by referencing them at the part component.

**Directives:**

Three kinds of Directives were made available for Angular 2:

Component Directives: They made components reusable by encapsulating logic in HTML, CSS, and JavaScript.

Decorator Directives: They can be used to decorate elements (for example, Hiding/Showing elements by ng-hide/ng-show or adding a tooltip).

Template Directives: These can turn HTML into a reusable template. The instantiating of this template and its insertion into the DOM could be completely controlled by the directive writer. Examples include ng-repeat and ng-if.

**Child Router:**

The Child router will convert every part of the program to a more compact application by supplying it with its own router. It helps to encapsulate the entire feature collections of a program.

**Screen Activator:**

With Angular 2, developers were able to take finer control on the navigation life cycle, through a set of can\* callbacks.

canActivate: It will allow or prevent navigation to the new control.

activate: It will respond to successful navigation to the new control.

canDeactivate: It will prevent or allow navigation away from the old controller.

deactivate: It will respond to successful navigation away from the old controller.

**Design:**

All this logic was built using a pipeline architecture that made it incredibly simple to add one's own actions into the pipeline or remove default ones. Moreover, its asynchronous character allowed developers to some make server requests to authenticate a user or load information for a control, while still in the pipeline.

**Logging:**

Angular 2.0 included a logging service known as diary.js--a very helpful attribute which measures where time is invested in your program (thus permitting you to identify bottlenecks in your code).

**Scope:**

$scope was removed from Angular 2.

**Angular 4 Features and Performance**

As compared to Angular 2, there are lots of new items added to this list. Not just new features but also some tweaks that improved old capabilities. So let's move on to see the list.

**Smaller and Faster:**

With Angular 4, programs will consume less space and run quicker than previous versions. And the staff is focused on continually making additional improvements.

**View Engine:**

They have made adjustments under to hood to exactly what AOT created code looks like. These modifications decrease the size of the generated code for those parts by approximately 60 percent. The more complicated the templates are, the greater the savings.

**Animation Package:**

They've pulled animations from the Angular core and set them in their own package. This means that in case you don't use animations, this excess code won't end up on your creation packages.

This feature will also enable you to easily find docs and to take advantage of auto-completion. You may add animations to the main NgModule by importing the Browser Animations Module out of @angular/platform-browser/animations.

**Improved \*ngIf and \*ngFor:**

The template binding syntax currently supports a few helpful alterations. Now you can utilize an if/else design syntax, and assign local variables like if to unroll an observable.

**Angular Universal:**

This release now contains the results of the external and internal work from the Universal team throughout the last few months. The vast majority of this Universal code is currently located in @angular/platform-server.

To learn more about using Angular Universal, have a look at the new renderModuleFactory method in @angular/platform-server, or Rob Wormald's Demo Repository. More documentation and code samples will come.

**TypeScript 2.1 and 2.2 Compatibility:**

The group has upgraded Angular into a more recent version of TypeScript. This will enhance the rate of ngc and you'll receive far better type checking during your program.

**Source Maps for Templates:**

Now whenever there's an error caused by something in one of the templates, they create source maps that provide a meaningful context concerning the original template.

**Conclusion:**

As I said earlier, Angular will be a bit confusing for those who are still in the learning phase. But for experienced developers who have knowledge of version 2, then it will be very easy for them to use and they will find it very helpful.

**What is JIT and AOT :**

NOTE : An Angular application consists mainly of components and their HTML templates. ... The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs that code.

Compiling your application during the build process provides a faster rendering in the browser.

NOTE :

**Angular offers two ways to compile your application:**

Just-in-Time (JIT), which compiles your app in the browser at runtime.

Ahead-of-Time (AOT), which compiles your app at build time.

**Q :** Javascript is intepreted language and Angular is compilled Lang How?

**A:**

**1.**Because a browser cannot **run the typescript code** and hence it needs to be compiled as vanilla JS. FYI, browsers do compile your JS.

**2.** Angular builds on Typescript, not Javascript.

**3.** Since Typescript is just a language Extension Browsers can't interpret it.

**4.** The Compiler (or more exact Transpiler) transpiles the Angular-Typescipt code to Javascript-Code that the browser can interpret.

*Q: Why compile with AOT?*

**Faster rendering**

With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.

**Fewer asynchronous requests**

The compiler inlines external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

**Smaller Angular framework download size**

There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

**Detect template errors earlier**

The AOT compiler detects and reports template binding errors during the build step before users can see them.

**Better security**

AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

**NOTE :**

**JIT and AOT in angular2**

If you are using angular2, the probability of you using TypeScript (TS) along with angular2 is very high. Since majority of the features of TS are still not supported by the browser, we might be relying on tools such as TypeScript compiler. These convert our code from TS to JS first, and then we serve these JS files to the browser (which again compiles the JS files to a binary which they understand).

**Why Use TypeScript?**

TypeScript is superior to its other counterparts like CoffeeScript and Dart programming languages in a way that TypeScript is extended JavaScript. In contrast, languages like Dart, CoffeeScript are new languages in themselves and require language-specific execution environment.

**The benefits of TypeScript include −**

Compilation − JavaScript is an interpreted language. Hence, it needs to be run to test that it is valid. It means you write all the codes just to find no output, in case there is an error. Hence, you have to spend hours trying to find bugs in the code. The TypeScript transpiler provides the error-checking feature. TypeScript will compile the code and generate compilation errors, if it finds some sort of syntax errors. This helps to highlight errors before the script is run.

Strong Static Typing − JavaScript is not strongly typed. TypeScript comes with an optional static typing and type inference system through the TLS (TypeScript Language Service). The type of a variable, declared with no type, may be inferred by the TLS based on its value.

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.

TypeScript supports Object Oriented Programming concepts like classes, interfaces, inheritance, etc.

**ES6 :**

ECMAScript, or ES6, was published in June 2015. It was subsequently renamed to ECMAScript 2015. Web browser support for the full language is not yet complete, though major portions are supported. Major web browsers support some features of ES6. However, it is possible to use software known as a transpiler to convert ES6 code into ES5, which is better supported on most browsers.

Let us now look at some major changes that ES6 brings to JavaScript.

**1. Constants**

Finally the concept of constants has made it to JavaScript! Constants are values that can be defined only once (per scope, scope explained below). A re-definition within the same scope triggers an error.

**Example :**

const JOE = 4.0

JOE= 3.5

// results in: Uncaught TypeError: Assignment to constant variable.

You can use the constant wherever you can use a variable (var).

console.log("Value is: " + joe \* 2)

// prints: 8

**2. Block-Scoped Variables and Functions**

Before this update, variables in JavaScript were function scoped. That is, when you needed a new scope for a variable, you had to declare it within a function.

Variables retain the value till the end of the block. After the block, the value in the outer block (if any) is restored.

{

let x = "hello";

{

let x = "world";

console.log("inner block, x = " + x);

}

console.log("outer block, x = " + x);

}

// prints

inner block, x = world

outer block, x = hello

**3. Arrow Functions**

ES6 brings a new syntax for defining functions using an arrow. In the following example, x is a function that accepts a parameter called a, and returns its increment:

var x = a => a + 1;

x(4) // returns 5

Using this syntax, you can define and pass arguments in functions with ease.

Using with a forEach():

[1, 2, 3, 4].forEach(a => console.log(a + " => " + a\*a))

// prints

1 => 1

2 => 4

3 => 9

4 => 16

**Define functions accepting multiple arguments by enclosing them in parentheses:**

[22, 98, 3, 44, 67].sort((a, b) => a - b)

// returns

[3, 22, 44, 67, 98]

**4. Default Function Parameters**

Function parameters can now be declared with default values. In the following, x is a function with two parameters a and b. The second parameter b is given a default value of 1.

var x = (a, b = 1) => a \* b

x(2)

// returns 2

x(2, 2)

// returns 4

Unlike other languages such as C++ or python, parameters with default values may appear before those without defaults. Note that this function is defined as a block with a return value by way of illustration.

var x = (a = 2, b) => { return a \* b }

However arguments are matched left to right. In the first invocation below, b has an undefined value even though a has been declared with a default value. The passed-in argument is matched with a rather than b. The function returns NaN.

x(2)

// returns NaN

x(1, 3)

// returns 3

When you explicitly pass in undefined as an argument, the default value is used if there is one.

x(undefined, 3)

// returns 6

**JavaScript | Rest Operator**

Rest Operator is an improved way to handle function parameter, allowing us to more easily handle various input as parameters in a function.

JavaScript has allowed a variable number of function parameters of a function but the problem is that it is not an array. It is an array like object. Therefore performing some operations on “arguments” will give an error.

Rest operator is added in ES2015 or ES6 which improved the ability to handle parameter.

**Syntax:**

function functionname[...parameters]//... is the rest operator

{

statement;

}

It stores n number of parameters as an array.

Code #1:

Javascript code for display parameter using rest operator

<script>

// Calling of function

function onlyMath(operator, ...numbers) {

document.write(operator +"<br>");

document.write(numbers);

}

onlyMath('Hello',1,2,3,4,5);

</script>

**Output**

Hello

1,2,3,4,5

**Code #2:**

Javascript code demonstrating addition of numbers using rest operator.

<script>

//function passed with parameters using rest operator.

function total(...args) {

var result = 0;

//args is used as a variable to add numbers.

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

result += args[i];

}

return result;

//scope of args ends here.

}

document.write(total(1, 6, 8));

</script>

**OutpuT:**

15

**6. String Templating**

String templating refers to interpolating variables and expressions into strings using a syntax like perl or the shell. A string template is enclosed in back-tick characters (`). By contrast single quotes (‘) or double quotes (“) indicate normal strings. Expressions inside the template are marked out between ${ and }. Here is an **example:**

var name = "joe";

var x = `hello ${name}`

// returns "hello joe"

Of course, you can use an arbitrary expression for evaluation.

// define an arrow function

var f = a => a \* 4

// set a parameter value

var v = 5

// and evaluate the function within the string template

var x = `hello ${f(v)}`

// returns "hello 20"

This syntax for defining strings can also be used to define multi-line strings.

var x = `hello world

next line`

// returns

hello world

next line

**7. Object Properties**

ES6 brings a simplified object creation syntax. Take a look at the example below:

var x = "hello world", y = 25

var a = { x, y }

// is equivalent to the ES5:

{x: x, y: y}

**// ES5-compatible code**

var myObject = {

prop1: 'hello',

prop2: 'world',

output: function() {

console.log(this.prop1 + ' ' + this.prop2);

}

};

myObject.output(); // hello world

// ES6 code

const lib = (() => {

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

function mult(a, b) { return a \* b; }

return { sum, mult };

}());

//console.log( lib.sum(2, 3) ); // 5

//console.log( lib.mult(2, 3) ); // 6

**8. Formal Class Definition Syntax**

**Class Definition**

And finally, JavaScript gets a formal class definition syntax. While it is merely syntactic sugar over the already available protytype-based classes, it does serve to enhance code clarity. That means this does not add a new object model or anything fancy like that.

class Circle {

constructor(radius) {

this.radius = radius

}

}

// use it

var c = new Circle(4)

// returns: Circle {radius: 4}

**Declaring Methods**

Defining a method is also quite simple. No suprises there.

class Circle {

constructor(radius) {

this.radius = radius

}

computeArea() { return Math.PI \* this.radius \* this.radius }

}

var c = new Circle(4)

c.computeArea()

// returns: 50.26548245743669

**Constructors**

**In ES5** or the current widely supported version of JavaScript, we use prototypes to create object inheritance. Before ES6, we used function constructors similar to this.

// ES5 Constructor Function

function Person(name) {

this.name = name;

}

var bob = new Person('Bob');

console.log(bob.name); // Outputs 'Bob'

**BUT**

**// ES2015/ES6 Class**

class Person {

constructor(name) {

this.name = name;

}

}

let bob = new Person('Bob');

console.log(bob.name); // Outputs 'Bob'

**Methods**

Next, let’s look at adding a function to our Person. In ES5 we would have had something like this.

// ES5 adding a method to the Person prototype

Person.prototype.walk = function() {

console.log(this.name + ' is walking.');

}

var bob = new Person('Bob');

bob.walk(); // Outputs 'Bob is walking.'

ES6 offers us a much more terse and clean syntax to achieve the same goal.

// ES6 Class adding a method to the Person prototype

class Person {

constructor(name) {

this.name = name;

}

walk() {

console.log(this.name + ' is walking.');

}

}

let bob = new Person('Bob');

console.log(bob.name); // Outputs 'Bob is walking'

**Get & Set**

ES6 classes brings a new syntax for getters and setters on object properties. Get and set allows us to run code on the reading or writing of a property. ES5 had getters and setters as well but was not widely used because of older IE browsers. ES5 getters and setters did not have as nice of a syntax that ES6 brings us. So let’s create a get and set for our name property.

// ES6 get and set

class Person {

constructor(name) {

this.\_name = name;

}

get name() {

return this.\_name.toUpperCase();

}

set name(newName) {

this.\_name = newName; // validation could be checked here such as only allowing non numerical values

}

walk() {

console.log(this.\_name + ' is walking.');

}

}

let bob = new Person('Bob');

console.log(bob.name); // Outputs 'BOB'

**Inheritance**

In addition to defining classes using the class keyword, you can also use the extends keyword to inherit from super classes. Let us see how this works with an example.

class Ellipse {

constructor(width, height) {

this.\_width = width;

this.\_height = height;

}

get area() { return Math.PI \* this.\_width \* this.\_height; }

set width(w) { this.\_width = w; }

set height(h) { this.\_height = h; }

}

class Circle extends Ellipse {

constructor(radius) {

super(radius, radius);

}

set radius(r) { super.width = r; super.height = r; }

}

// create a circle

var c = new Circle(4)

// returns: Circle {\_width: 4, \_height: 4}

c.radius = 2

// c is now: Circle {\_width: 2, \_height: 2}

c.area

// returns: 12.566370614359172

c.radius = 5

c.area

// returns: 78.53981633974483

**Template Tag :**

You may already be familiar with ES6 **template literals**, which allows string interpolation like this:

const name = 'Steve';

const message = `Hello ${name}!`;

console.log(message); // Output -> Hello Steve!

**ES6 is basically EcmaScript 6 / EcmaScript 2015.**

EcmaScript is not any scripting language instead a standard that Javascript is based upon. So, ES6 is a new version or new standard of Javascript.

ES6 brings many new feature like concept of classes, template tags, arrow functions etc.

Almost all the modern browsers support ES6 but for the old browsers there are many transpilers e.g. Babel.js those we need to include at top of our code to transpile ES6 to ES5 (Javascript with old standards).

All of the popular javascript libraries and frameworks like Node.js, ReactJS follow ES6.

**ES6 or EcmaScript 2015 is a significant update to JavaScript with many useful features:**

* Arrow functions
* constant keyword (value cannot be changed)
* let keyword (scope level)
* Destructuring
* Classes
* Many Higher-order functions
* Template strings
* default value in function parameter
* rest parameter
* spread operator
* generators
* promises
* modules (import/export)
* and many more….
* Before ES6 we had to use Other libraries such as UnderscoreJS or LodashJS

Many of the ES6 features are not supported in most of the browsers, so use Babel (It compilers your ES6 code into ES5 code which is supported in almost all browsers).

**The Spread Syntax**

The spread syntax is simply **three dots: ...**

It allows an iterable to expand in places where 0+ arguments are expected.

Definitions are tough without context. Lets explore some different use cases to help understand what this means.

**Example #1 — Inserting Arrays**

Take a look at the code below. In this code, we don’t use the spread syntax:

var mid = [3, 4];

var arr = [1, 2, mid, 5, 6];

console.log(arr);

O/P :

[1, 2, [3, 4], 5, 6]

**Is that the result you expected?**

By inserting the mid array into the arr array, we’ve ended up with an array within an array. That’s fine if that was the goal, but what if want only a single array with the values of 1 through 6? To accomplish this, we can use the spread syntax! Remember, the spread syntax allows the elements of our array to expand.

Lets look at the code below. Everything is the same — except we’re now using the spread syntax to insert the mid array into the arr array:

var mid = [3, 4];

var arr = [1, 2, **...mid**, 5, 6];

console.log(arr);

// [1, 2, 3, 4, 5, 6]

**Example #2 — Math**

JavaScript has a built in math object that allows us to do some fun math calculations. In this example we’ll be looking at Math.max(). If you’re unfamiliar, Math.max() returns the largest of zero or more numbers. Here are a few examples:

Math.max();

// -Infinity

Math.max(1, 2, 3);

// 3

Math.max(100, 3, 4);

// 100

As you can see, if you want to find the maximum value of multiple numbers, Math.max() requires multiple parameters. You unfortunately can’t simply use a single array as input. Before the spread syntax, the easiest way to use Math.max() on an array is to use .apply()

var arr = [2, 4, 8, 6, 0];

function max(arr) {

return Math.max.apply(null, arr);

}

console.log(max(arr));

**// 8**

Now take a look at how we do the same exact thing with the spread syntax:

var arr = [2, 4, 8, 6, 0];

var max = Math.max(...arr);

console.log(max);

**// 8**

**Example #3 — Copy an Array**

var arr = ['a', 'b', 'c'];

var arr2 = arr;

arr2.push('d');

console.log(arr);

// ['a', 'b', 'c', 'd']

No need to fear though! We can use the spread operator!

Consider the code below. It’s almost the same as above. Instead though, we’ve used the spread operator within a pair of square brackets:

var arr = ['a', 'b', 'c'];

var arr2 = [...arr];

console.log(arr2);

Hit run, and you’ll see the expected output:

['a', 'b', 'c']

Above, the array values in arr expanded to become individual elements which were then assigned to arr2. We can now change the arr2 array as much as we’d like with no consequences on the original arr array:

var arr = ['a', 'b', 'c'];

var arr2 = [...arr];

arr2.push('d');

console.log(arr);

**String to Array**

As a fun final example, you can use the spread syntax to convert a string into an array. Simply use the spread syntax within a pair of square brackets:

var str = "hello";

var chars = [...str];

console.log(chars);

// ["h", "e", "l", "l", "o"]

**Rest parameters ...**

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

**Like here:**

function sum(a, b) {

return a + b;

}

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

**Ex2:**

function sumAll(...args) { // args is the name for the array

let sum = 0;

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

return sum;

}

alert( sumAll(1) ); // 1

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

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

**Ex3:**

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)

**Ex4 :**

We can even combine the spread operator with normal values:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

**Type Inference**

var a = "some text";

var b = 123;

a = b; // Compiler Error: Type 'number' is not assignable to type 'string'

**ex1:**

var arr = [0, 1, "test"];

arr.push("str");

The compiler accepts the new value since the new value is of type string which is okay.

Now, lets try to add a new type to the array which was not already a part of the array:

var arr = [0, 1, "test"];

arr.push("str") // OK

arr.push(true);

// Compiler Error: Argument of type 'true' is not assignable to parameter of type 'string | number'.

The above code will show a compiler error because boolean is not a part of union (string | number).

**Ex2 :**

function sum(a: number, b: number )

{

return a + b;

}

var total: number = sum(10,20); // OK

var str: string = sum(10,20); // Compiler Error

**TypeScript - Function Overloading:**

TypeScript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.

**Example: Function Overloading:**

function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {

return a + b;

}

add("Hello ", "Steve"); // returns "Hello Steve"

add(10, 20); // returns 30

**Example: Function Overloading -- Error Code**

function display(a:string, b:string):void //Compiler Error: Duplicate function implementation

{

console.log(a + b);

}

function display(a:number): void //Compiler Error: Duplicate function implementation

{

console.log(a);

}

**Type Inference in TypeScript:**

**Variable Definition**

Types of a variable are inferred by definition.

let foo = 123; // foo is a `number`

let bar = "Hello"; // bar is a `string`

foo = bar; // Error: cannot assign `string` to a `number`

**Static functions**

You can use the static keyword to make class methods static. A static method acts on the class itself, not on instances of the class:

class Person{

constructor(name, age) {

this.name = name

this.age = age

}

static describe(){

console.log("This is a person.")

}

sayName() {

console.log("My name is " + this.name)

}

}

Person.describe()

//logs 'This is a person.'

Notice how static methods operate on the class itself and not an instance of the class. We didn't have to create a new Person to call the static method.

Static methods are useful for common or shared class functionality. In this case, the describe() method is used to describe what the Person class is. It will apply to every instance of Person. This is why we make it a static method.

**Class Inheritance**

Inheritance allows you to create new classes based off existing ones. These new classes "inherit" the methods and properties of their parent. They can also override or extend the parent:

class Person{

constructor(name, age) {

this.name = name

this.age = age

}

static describe(){

console.log("This is a person.")

}

sayName() {

console.log("My name is " + this.name)

}

}

class Programmer extends Person {

sayName(){

console.log("My name is " + this.name + " and I am a programmer!")

}

}

let averageJoe = new Person('Todd', 40)

let programmer = new Programmer('Sam', 33)

averageJoe.sayName()

//logs 'My name is Todd'

programmer.sayName()

//logs 'My name is Sam and I am a programmer!'

Using the extends keyword, we can create a new class sharing the same characteristics as Person. Notice how we override the sayName() method with a new definition for the Programmer class. Apart from overriding this method, everything else remains the same for both Person and Programmer.

**Using super**

The super keyword allows a child class to invoke parent class properties and methods.

class Person{

constructor(name, age) {

this.name = name

this.age = age

}

static describe(){

console.log("This is a person.")

}

sayName() {

console.log("My name is " + this.name)

}

}

class Programmer extends Person {

sayName(){

super.sayName()

console.log("My name is " + this.name + " and I am a programmer!")

}

}

let averageJoe = new Person('Todd', 40)

let programmer = new Programmer('Sam', 33)

programmer.sayName()

//logs 'My name is Sam'

//logs 'My name is Sam and I am a programmer!'

Notice how we call super.sayName() in the Programmer implementation of sayName(). While this invokes the parent implementation of super.sayName(), the name property still references the Programmer class.

**Generator functions vs regular functions:**

Regular functions run to completion. There is no pause or break during execution. With generator functions, you can pause execution and pass in new values. By using the yield keyword, a generator function can be paused, passed new values, and resumed at a later time.

**Syntax**

Generator functions have almost identical syntax to regular functions. The only real difference is the \* used in the definition:

function \*myGen(){

//valid

}

function\* myGen(){

//valid

}

function\*myGen(){

//valid

}

**How generator functions work:**

Generator functions use iterators to pause and resume functions. Generators use the yield keyword to pause execution internally. When a generator function comes across yield, it returns an iterator object like this:

{value: x, done: bool}

This iterator object has a property value with the current yielded value and done indicating the completion of the iteration object:

function \*myGen() {

let a = yield 'the first yield returned'

console.log(a)

let b = yield 'the second yield returned'

console.log(b)

return "all done"

}

let generator = myGen()

generator.next()

//returns {value: 'the first yield returned', done: false}

generator.next("hi")

//logs 'hi'

//returns {value: 'the second yield returned', done: false}

generator.next("bye")

//logs 'bye'

//returns {value: 'all done', done: true}

Notice how we first define a basic generator function \*myGen with two yield statements. We then assign the variable generator the invoked generator function myGen().

Whenever we call next(), the function resumes until it hits the next yield clause. Notice how the first next() call returns the first yield returned. When we call next() a second time, we pass in the argument "hey" and resume execution. Notice how our passed value "hey" is then logged before the second yield is encountered. After calling next() again, the function finds no more yields and returns.

**Using generators in the real world:**

Generators are great for working with asynchronous operations in JavaScript. They work especially well with promises to handle async activity. Specifically, you can use generators to prevent excessive promise chaining:

const makeRequest = () => new Promise((resolve) => {

setTimeout(() => resolve('success'), 500)

})

const myGenerator = function\* () {

yield

const response = yield makeRequest()

// waits for promise

yield response

}

Notice how our myGenerator function waits for the promise in makeRequest() to resolve before continuing. While we could have used then() to handle the resolved promise, this syntax eliminates the need for nested chaining and response handling.

In the real world, we would need to call next() to progress through the function. Ideally, we would want the myGenerator() function to resolve without having to call next() for each yield. Thankfully, libraries like Co.js implement wrappers that resolve generators automatically to optimize them for real world scenarios.

----------------------------------------------------------------------------------------

**What is asynchronous programming?**

Asynchronous programming involves running a process separately from a main thread and notifying the main thread when it completes. This concept is better explained through a real world scenario:

Let's say your web app returns a list of items when a button is clicked. When the user clicks the button, an HTTP GET request is made to a server to retrieve the list of items. This means the user waits for network communication between the client and the server to take place before he/she sees the list of items.

When the HTTP request is made asynchronously the user can still navigate the web app while the information loads. If the HTTP request is made synchronously, the user has to wait for the response to continue using the app.

This is the key advantage of asynchronous programming. It allows you to run potentially blocking I/O operations in the background without interrupting a main thread or process.

**How does a Promise work?**

The Promise objects represents the eventual completion of an asynchronous operation. It returns a single value based on an operation being resolved or rejected. A promise is always in one of three stages:

* fulfilled
* rejected
* pending

The Promise() constructor takes two arguments: a resolve function and a reject function. It returns one or the other based on the outcome of the asynchronous operation.

**Creating a Promise**

You can create a promise in JavaScript using the Promise constructor:

const myPromise = amount => {

return new Promise((resolve,reject) => {

if(amount > 0){

resolve("success!")

}

reject("failure!")

})

}

myPromise(1)

//resolves Promise { 'success!' }

In the example above, we create a myPromise() function that takes a single amount parameter and returns a Promise object. The promise constructor function takes two arguments: resolve and reject.

We then call our myPromise(1) function. Notice how it returns a resolved Promise object with the "success!" message since our argument 1 is greater than 0.

Promise Methods

Promise methods exist to handle the resolution or rejection of a Promise object. Below is a brief description and example of the methods commonly used with JavaScript promises:

then()

The then() method executes after a promise is either fulfilled or rejected. It takes two function arguments for resolved and rejected.

let handleSuccess = (x) => {

console.log(x + " it worked!")

}

let handleError = (x) => {

console.log(x + " oh no, it failed!")

}

const myPromise = amount => {

return new Promise((resolve,reject) => {

if(amount > 0){

resolve("success!")

}

reject("failure!")

})

}

myPromise(1).then(handleSuccess, handleError)

//logs 'success! it worked!'

myPromise(0).then(handleSuccess,handleError)

//logs 'failure! oh no, it failed!'

Notice how we call then() twice on our constructed myPromise() function. It's important to remember that the method accepts two functions as arguments, in our case handleSuccess() and handleError(). Notice how the original message gets passed to the handler function for both resolved and rejected scenarios.

catch()

The catch() method provides a better way to handle rejections and failures. It is a "catch all" for any rejected promise:

const myPromise = amount => {

return new Promise((resolve,reject) => {

if(amount > 0){

resolve("success!")

}

reject("failure!")

})

}

myPromise(0).then(res => {

console.log(res + " success!")

}).catch(err => {

console.log(err + "oh no, it failed!")

})

//logs 'failure! oh no, it failed!'

In the above example, catch() takes the returned promise from the then() function and handles the rejection with its own argument function. Notice how we only pass a single argument to the then() function for handling a successful response.

Not only does the catch() method save us from having to specify the second argument for then(), it also catches any other errors along the way. Any internal errors thrown by then() will still be caught by catch():

const myPromise = amount => {

return new Promise((resolve,reject) => {

if(amount > 0){

resolve("success!")

}

reject("failure!")

})

}

myPromise(1).then(res => {

throw new Error();

console.log(res + " success!")

}).catch(err => {

console.log(err + "oh no, it failed!")

})

//logs 'Error oh no, it failed!'

Even if we throw an error within our then() handler, the catch() method will still catch it!

Promise.resolve()

Returns a resolved promise with the given value:

Promise.resolve("Success")

//returns Promise {'Success'}

Promise.reject()

Returns a rejected promise with the given value:

Promise.reject("error")

//returns unhandled promise rejection

Promise.all()

The all() method takes an array of promises as an argument. It returns a resolved promise based on the referenced promises ALL being fulfilled or rejected:

const p1 = new Promise((resolve,reject) => {

setTimeout(resolve("p1 success"),2000)

})

const p2= new Promise((resolve,reject) =>{

setTimeout(resolve("p2 success"),4000)

})

Promise.all([p1,p2]).then(res => {

console.log(res);

})

//logs ['p1 success', 'p2 success'] after 4 seconds

Promise.race()

The race() method takes an array of promises as an arugment. It returns a resolved promise based on the first referenced promise that resolves.

const p1 = new Promise((resolve,reject) => {

setTimeout(resolve("p1 success"),2000)

})

const p2= new Promise((resolve,reject) =>{

setTimeout(resolve("p2 success"),4000)

})

Promise.race([p1,p2]).then(res => {

console.log(res);

})

//logs 'p1 success' since p1 finishes first

Since p1 is the first to resolve, race() returns the resolved promise for p1 .

**TypeScript - Tuples**

At times, there might be a need to store a collection of values of varied types. Arrays will not serve this purpose. TypeScript gives us a data type called tuple that helps to achieve such a purpose.

It represents a heterogeneous collection of values. In other words, tuples enable storing multiple fields of different types. Tuples can also be passed as parameters to functions.

**Syntax**

var tuple\_name = [value1,value2,value3,…value n]

For Example

var mytuple = [10,"Hello"];

You can also declare an empty tuple in Typescript and choose to initialize it later.

var mytuple = [];

mytuple[0] = 120

mytuple[1] = 234

**Accessing values in Tuples**

Tuple values are individually called items. Tuples are index based. This means that items in a tuple can be accessed using their corresponding numeric index. Tuple item’s index starts from zero and extends up to n-1(where n is the tuple’s size).

**Syntax**

tuple\_name[index]

Example: Simple Tuple

var mytuple = [10,"Hello"]; //create a tuple

console.log(mytuple[0])

console.log(mytuple[1])

In the above example, a tuple, mytuple, is declared. The tuple contains values of numeric and string types respectively.

On compiling, it will generate the same code in JavaScript.

Its output is as follows −

10

Hello

**Never**

A video lesson on the never type

Programming language design does have a concept of bottom type that is a natural outcome as soon as you do code flow analysis. TypeScript does code flow analysis (😎) and so it needs to reliably represent stuff that might never happen.

The never type is used in TypeScript to denote this bottom type. Cases when it occurs naturally:

A function never returns (e.g. if the function body has while(true){})

A function always throws (e.g. in function foo(){throw new Error('Not Implemented')} the return type of foo is never)

Of course you can use this annotation yourself as well

let foo: never; // Okay

However, only never can be assigned to another never. e.g.

let foo: never = 123; // Error: Type number is not assignable to never

// Okay as the function's return type is `never`

let bar: never = (() => { throw new Error('Throw my hands in the air like I just dont care') })();

Great. Now let's just jump into its key use case :)

Use case: Exhaustive Checks

You can call never functions in a never context.

function foo(x: string | number): boolean {

if (typeof x === "string") {

return true;

} else if (typeof x === "number") {

return false;

}

// Without a never type we would error :

// - Not all code paths return a value (strict null checks)

// - Or Unreachable code detected

// But because TypeScript understands that `fail` function returns `never`

// It can allow you to call it as you might be using it for runtime safety / exhaustive checks.

return fail("Unexhaustive!");

}

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

And because never is only assignable to another never you can use it for compile time exhaustive checks as well. This is covered in the discriminated union section.