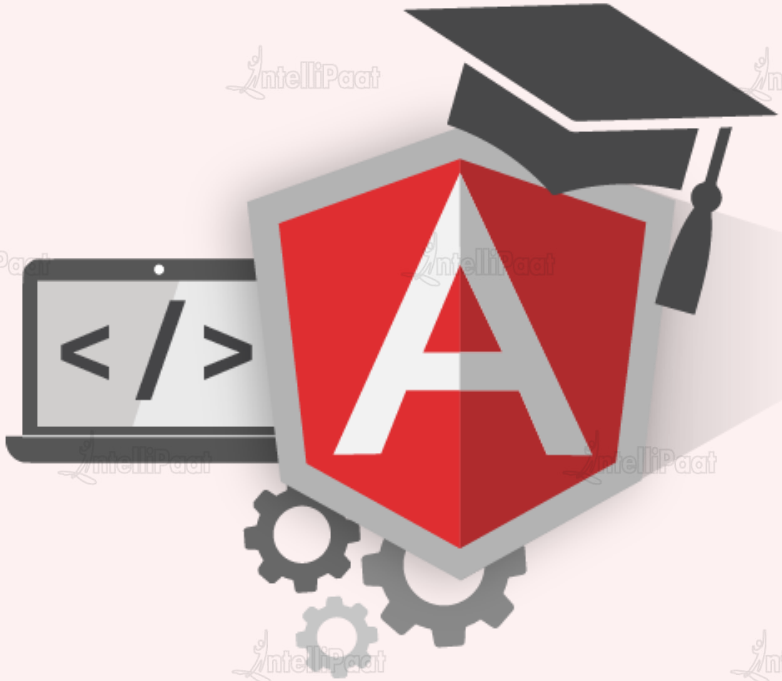
IntelliPaat

Angular

Module - TypeScript



Agenda

01 Overview of Typescript

02 Installation

03 Data Types

04 Variable Declaration & Various Scopes

05 Function Declaration & Function Expression

06 Let & Const

07 Destructuring

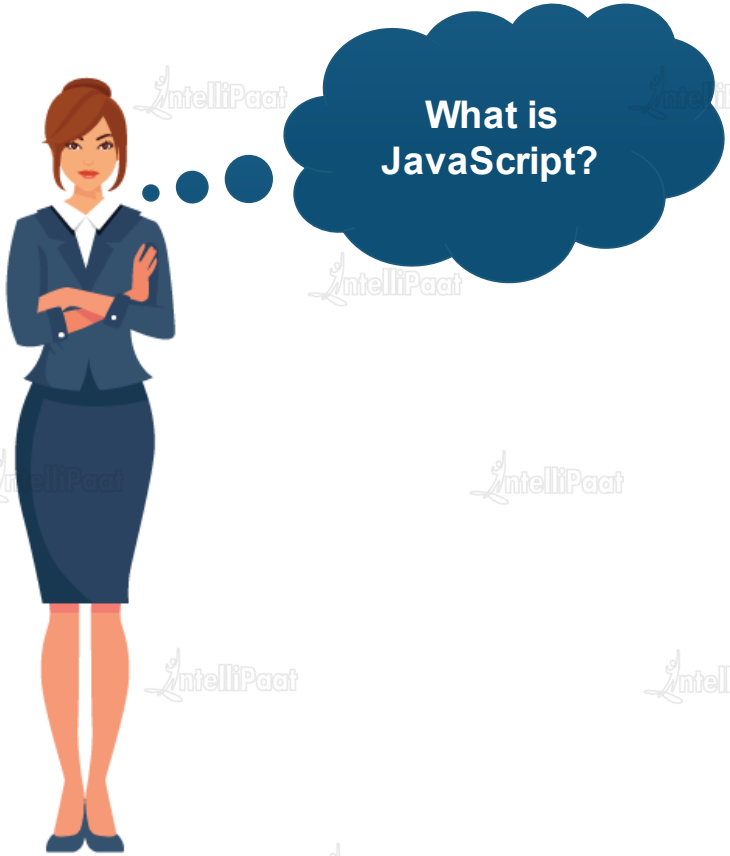
08 Classes

09 Interfaces

10 Maps & Sets

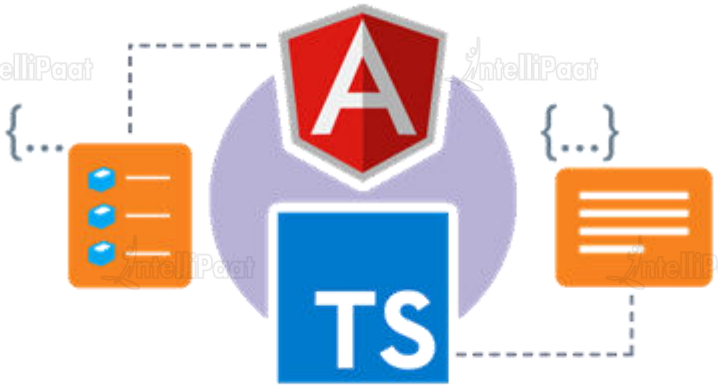
Overview of TypeScript

JavaScript



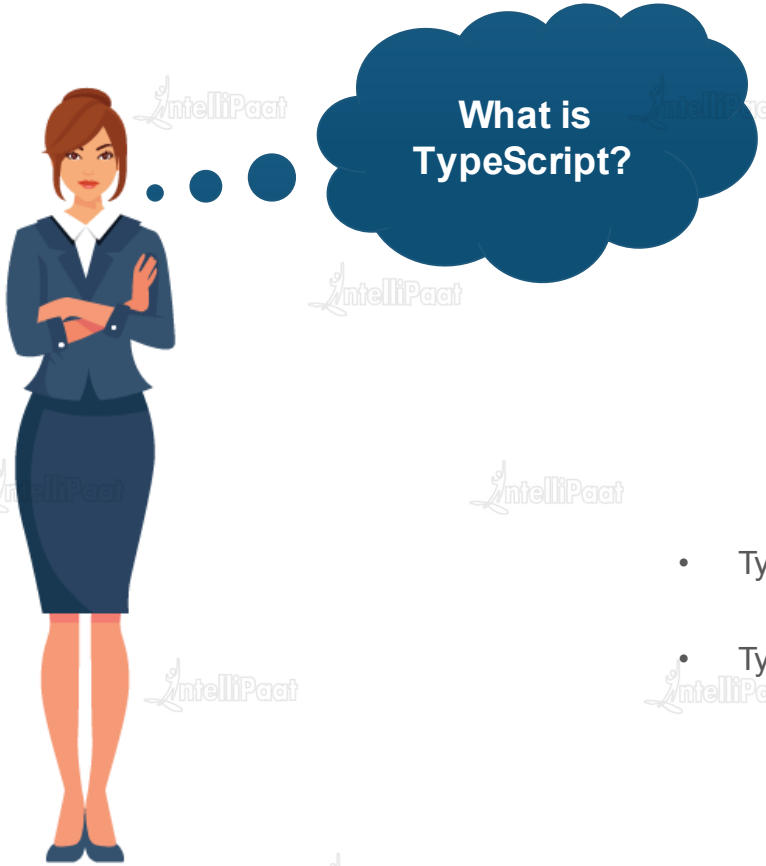
- JavaScript was introduced as a language for the client side.
- The development of Node.js has marked JavaScript as an emerging server-side technology too.
- JavaScript is **interpreted** and **loosely typed language**.

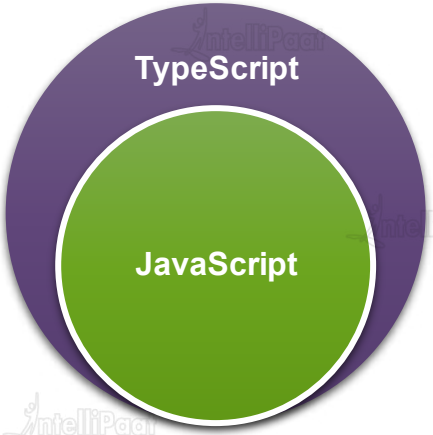
Need for TypeScript



TypeScript is presented to overcome the drawback.

TypeScript

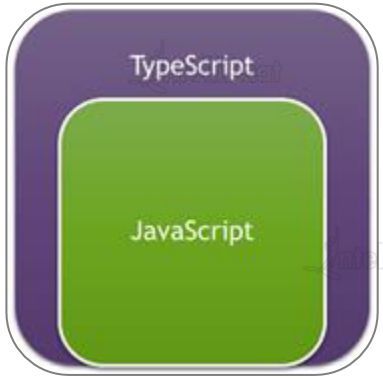




- TypeScript is JavaScript plus some additional features.
- TypeScript is a strongly typed, object oriented, compiled language.

https://www.typescriptlang.org

Features of TypeScript



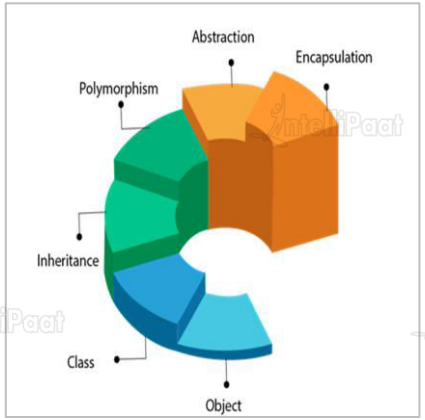
TypeScript is just JavaScript Code

TypeScript supports other JS libraries

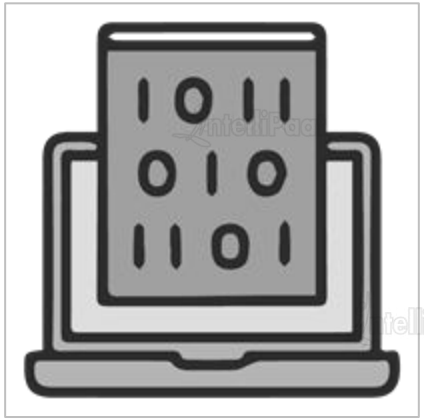
JavaScript files can be renamed as TypeScript

TypeScript is Portable

Benefits of TypeScript



Supports Object Oriented Programming



Compilation using Typescript Compiler or Transpiler

```
var
str:string =
123;
```

Strongly typed language

Installation

Environment Setup

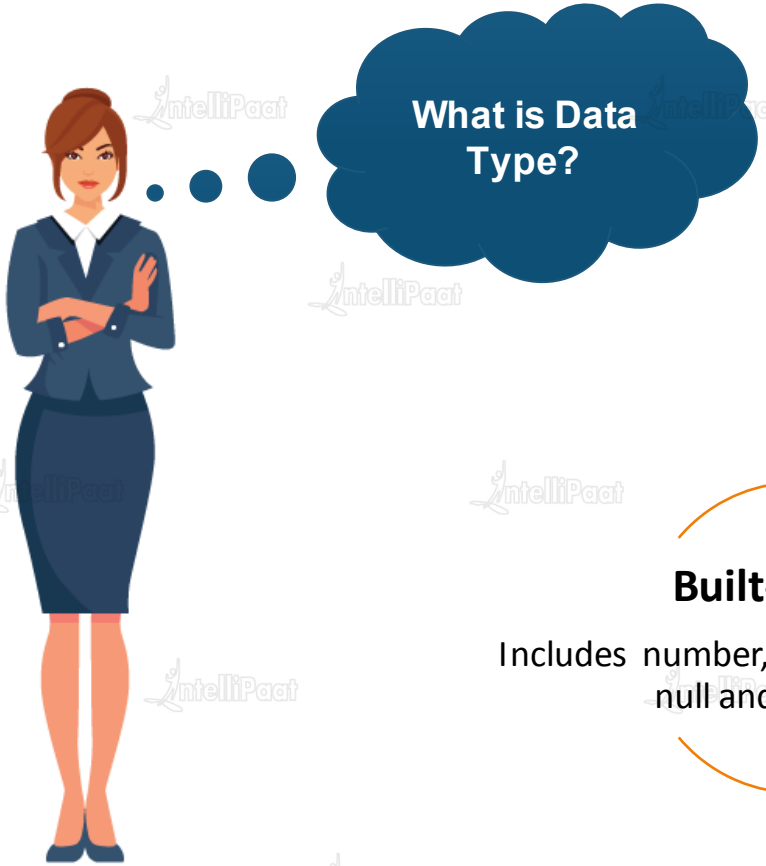
Steps

- Install nodejs
- Intstall typescript by giving command

  npm install –g typescript

Data Types

Data Types



any

The super type of all data types

Built-in Type

Includes number, string, boolean, void, null and undefined

User-defined Type

Include Array, Enums, Classes, Interfaces, etc.

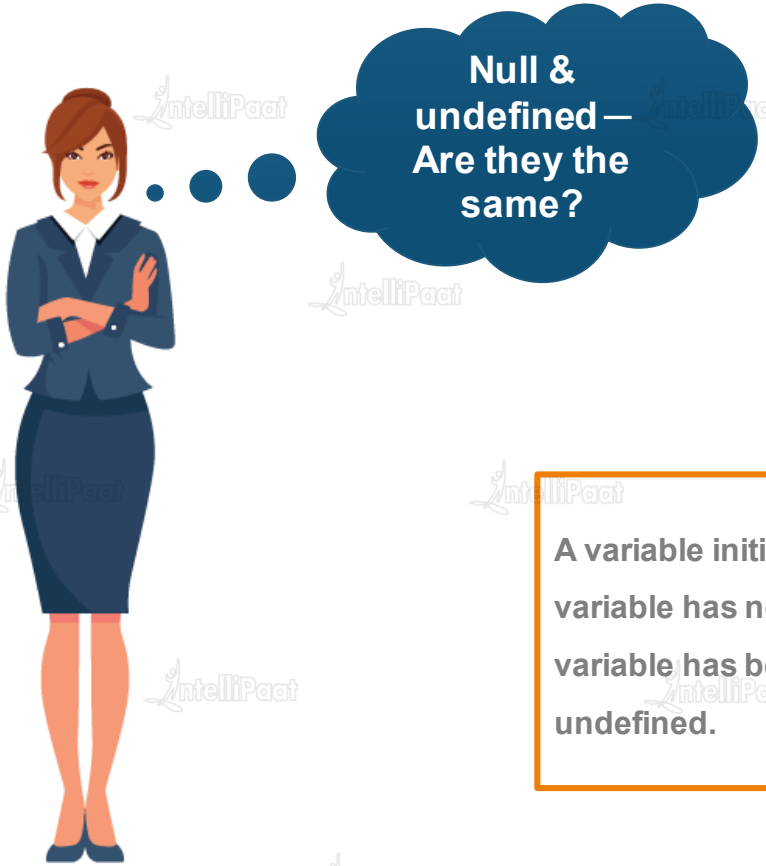
any Data Types

- The any data type is the supertype of all types in TypeScript.
- It denotes a dynamic type.
- Using the any type is equivalent to opting out of type checking for a variable.

Built in Data Type

Data Type	Keyword	Description
Number	number	Double precision 64-bit floating used to represent both, integers and fractions.
Boolean	boolean	Represents logical values true and false
String	string	Used to represent a sequence of Unicode characters
Void	void	Used on function return-types to represent non returning functions
Null	null	Represents an intentional absence of an object value
Undefined	undefined	Denotes value given to uninitialized variable

Null and undefined Data Type



The null and undefined cannot be used to reference the data type of a variable. They can only be assigned as values to a variable.

A variable initialized with undefined means that the variable has no value while null means that the variable has been set to an object whose value is undefined.

Use defined Data Type

- Include
  - Enumerations (enums)
  - Classes
  - Interfaces
  - Arrays

Variable Declaration & Various Scopes

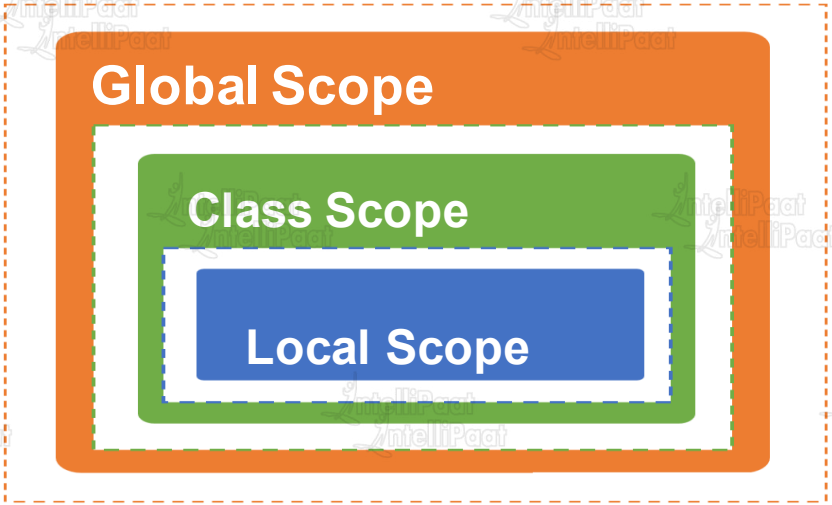
Variable Declaration

When you declare a variable, you have four options -

Declare its type and value in one statement.	Declare its type but no value. In this case, the variable will be set to undefined.	Declare its value but no type. The variable's type is inferred from the data type of the value.	Declare neither value not type. In this case, the data type of the variable will be any and will be initialized to undefined.
var name:string = "mary"	var name:string;	var name = "mary"	var name;

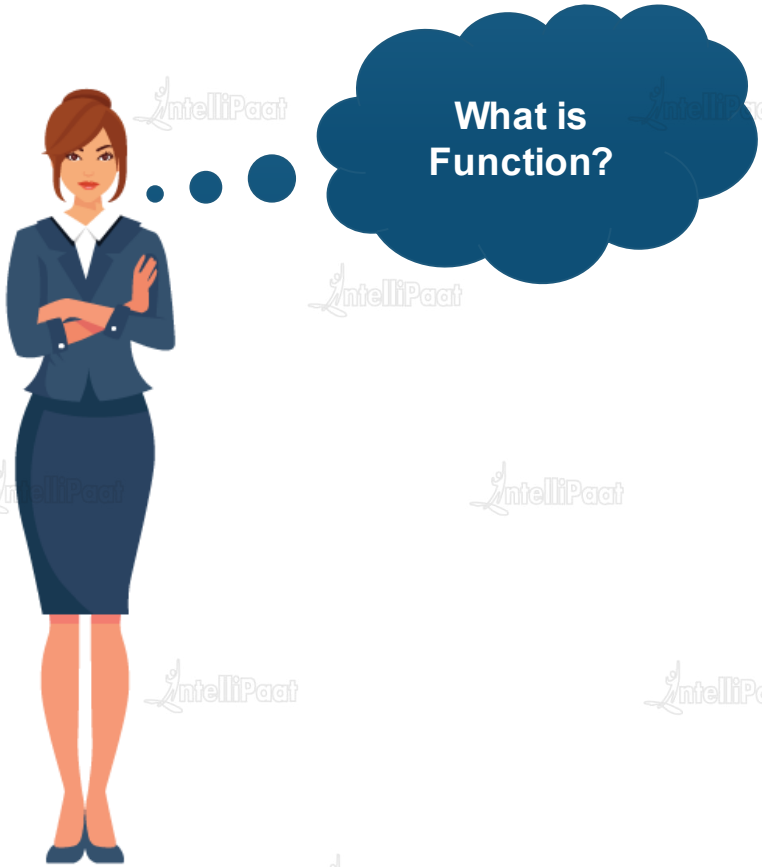
Variable Scopes

The scope of a variable specifies where the variable is defined.



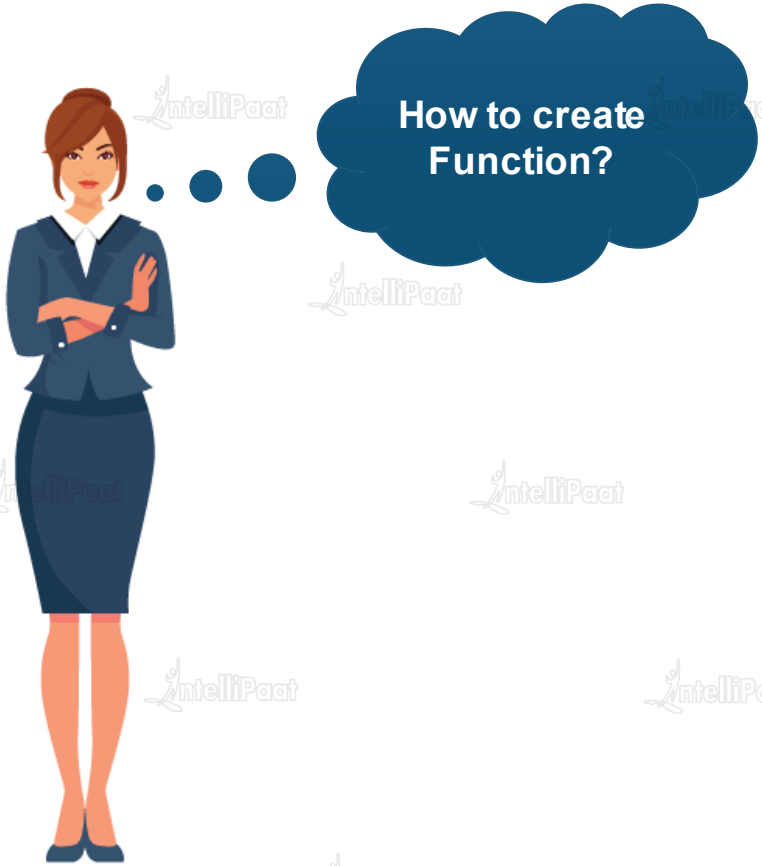
Function Declaration & Function Expression

Function



- Fundamental building blocks in JavaScript.
- A set of statements that performs a task or calculates a value
- To use a function, you must define it somewhere in the scope from which you wish to call it.

Function Creation

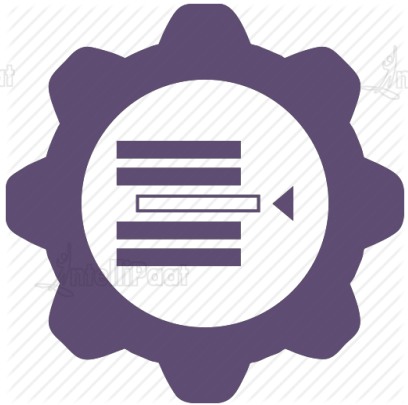


- Function Expression
- Function Declaration
- Function Constructor

Function Expression & Function Declaration

- **Function Declaration**

  function foo() { ... }



- **Function Expression**

  a) **Named Function Expression**

  var foo = function bar() { ... }

  b) **Anonymous Function Expression**

  var foo = function() { ... }

  c) **Immediately Invoked Function Expression (IIFE)**

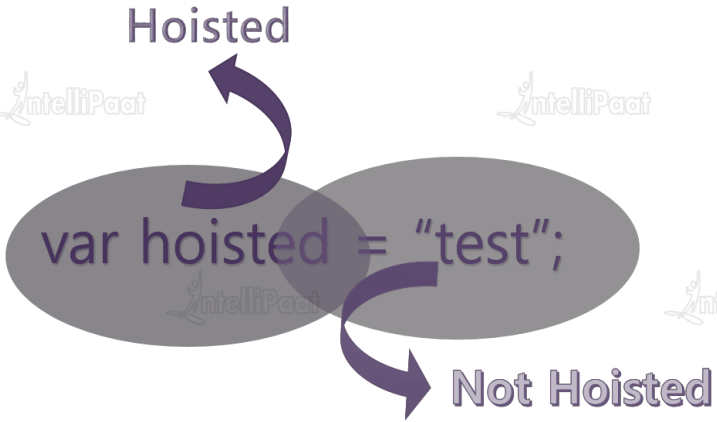
  (function() { ... }());

Difference between Function Expression & Function Declaration

- Function expression and function declaration are not synonymous.
- Unlike a function expression, a function declaration is bound by the function name.
- The fundamental difference between the two is that, function declarations are parsed before their execution.
- On the other hand, function expressions are parsed only when the script engine encounters it during execution.

Hoisting

Function Declarations and Function Variables are always moved ('hoisted') to the top of their JavaScript scope by the JavaScript interpreter but function expressions are never hoisted.



Function Hoisting

Hoisting difference between Function Expression & Function Declaration

```
hoisted(); // logs "foo"

function hoisted()

{ console.log('foo'); }
```

```
notHoisted(); // TypeError: notHoisted is not a function

var notHoisted = function()

{ console.log('bar'); };
```

Function Constructor

TypeScript also supports defining a function with the built-in JavaScript constructor called Function()

Syntax

```
var res = new Function( [arguments] ) { ... }
```

Example

```
var myFunction = new Function("a", "b", "return a * b");

var x = myFunction(4, 3);

console.log(x);
```

Output

12

IIFE

An IIFE (**Immediately Invoked Function Expression**) is a JaveScript function or a self excecuting Anonymous function that runs as soon as it is defined.

Syntax

```
(function () { statements })();
```

Let & const

let and const

Block-scoped binding constructs. let is the new var and const is single-assignment.

Demo

Difference b/w let and var

Example:

```
var a = 1;
 var b = 2;
if (a === 1)
{ var a = 11; // the scope is global
let b = 22; // the scope is inside the if-block
console.log(a); // 11
 console.log(b); // 22 }
console.log(a); // 11
console.log(b); // 2
```

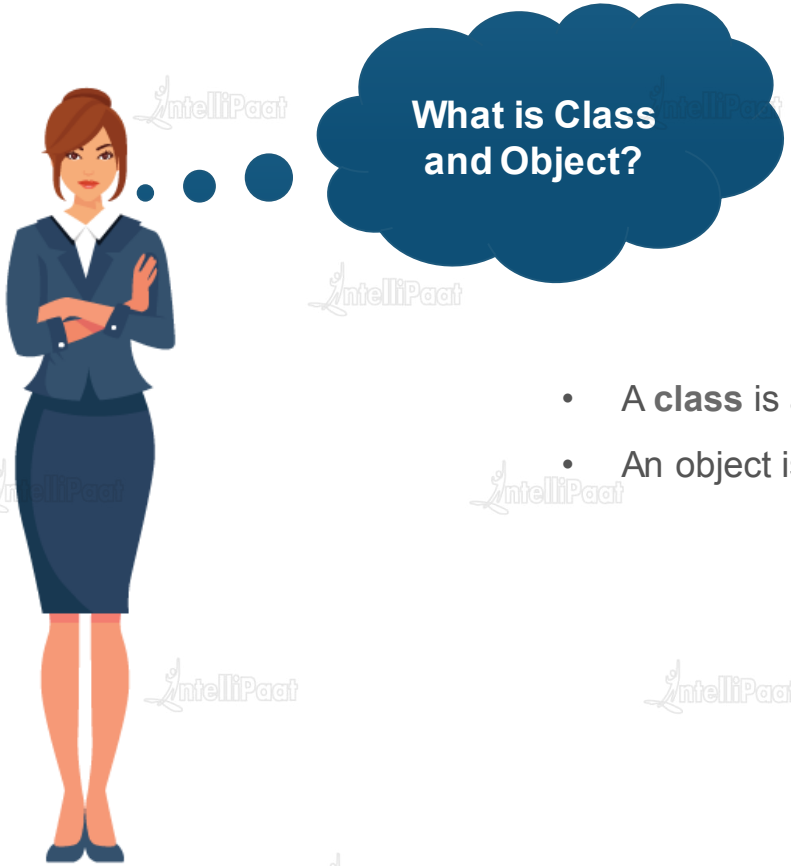
Difference b/w let and var

At the top level of programs and functions, let, unlike var, does not create a property on the global object.

Demo

Class

Class



- A **class** is a blueprint that contains the variables and the methods.
- An object is an instance of class.

Class Components

What does Class contain?

Any variable declared in a class. Fields represent data pertaining to objects

Fields/Data members

Responsible for allocating memory for the objects of the class

Constructors

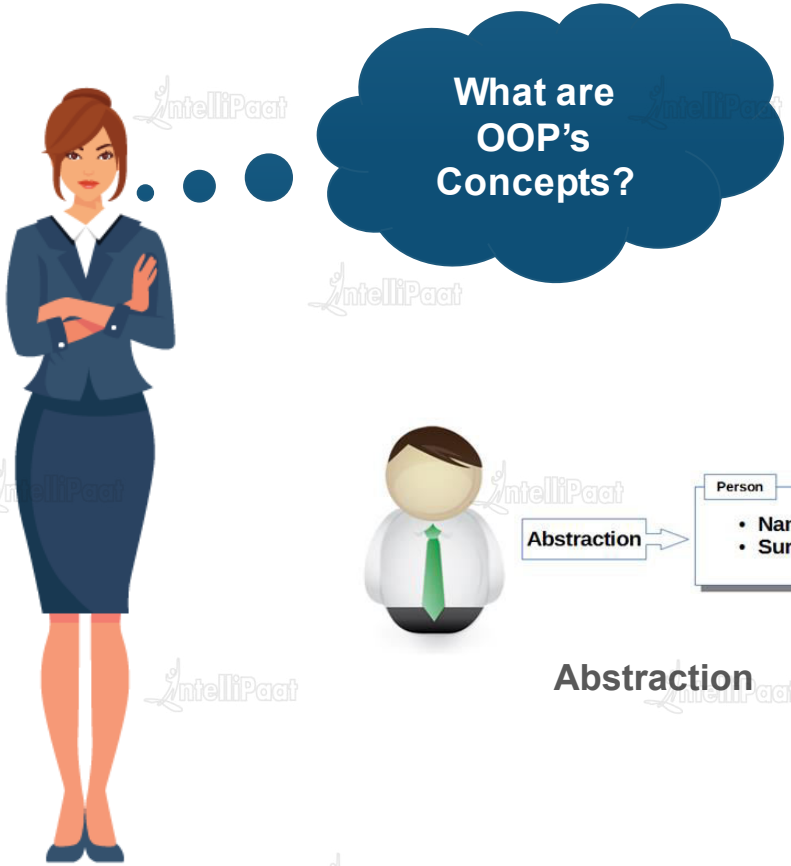
Represent actions an object can take. They are also at times referred to as methods

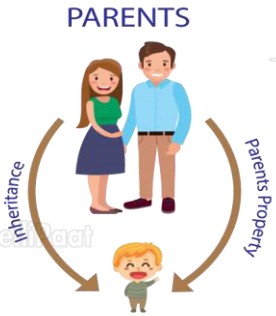
Member function

How to create a Class?

OOP's Concepts

OOP's Concepts

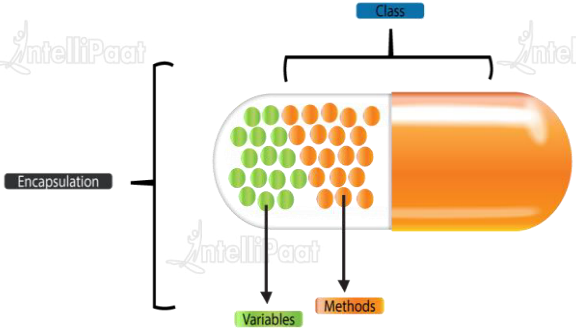




Inheritence

Polymorphism

Abstraction

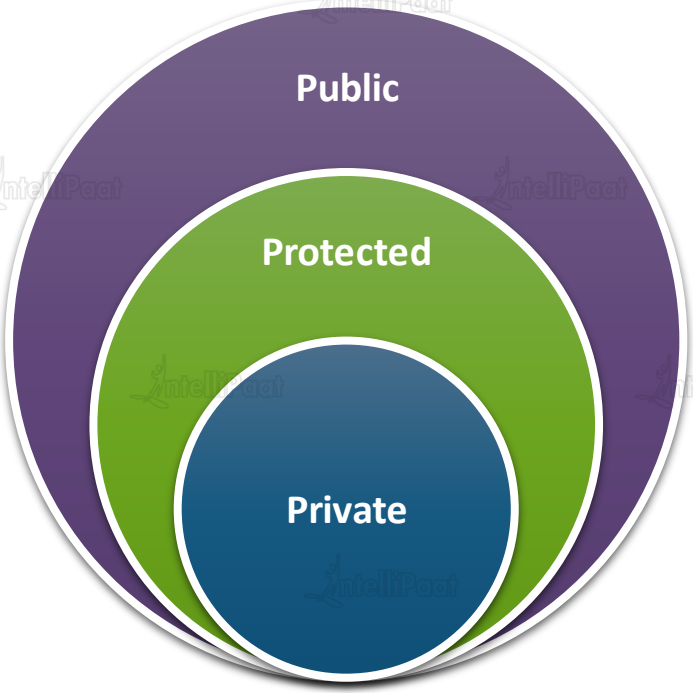


Encapsulation (Data Hiding)

Encapsulation(Data Hiding)

- A class can control the visibility of its data members to members of other classes. This capability is termed as Data Hiding or Encapsulation.
- Object Orientation uses the concept of access modifiers to implement the concept of Encapsulation. The access specifiers/modifiers define the visibility of a class's data members outside its defining class.

Types of Access Modifiers



Inheritance

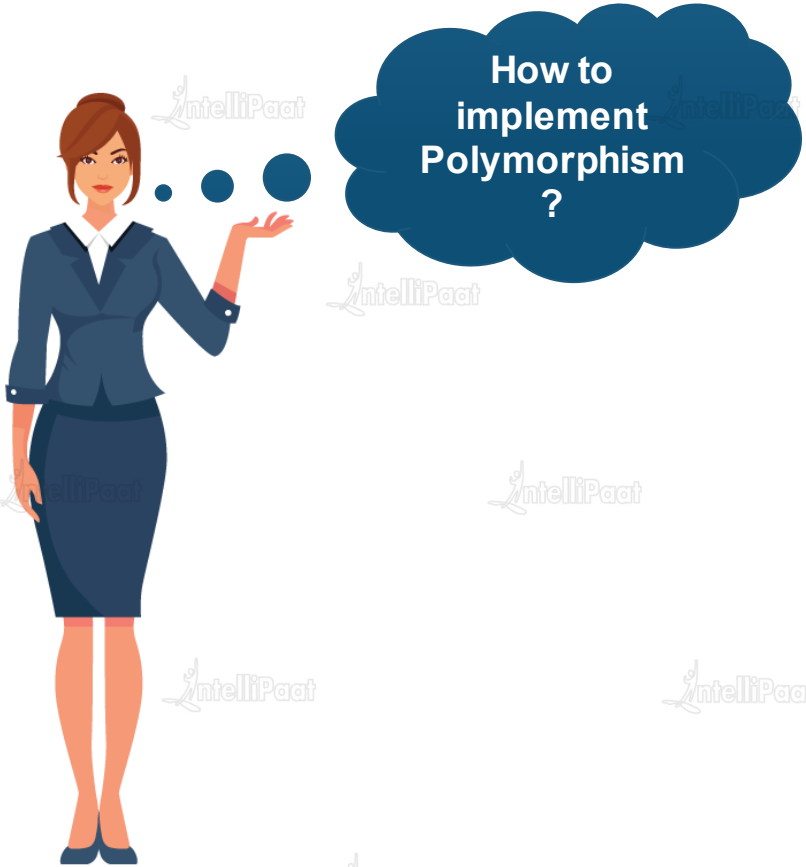
- TypeScript supports the concept of Inheritance.
- Inheritance is the ability of a program to create new classes from an existing class.

Inheritance Implementation

Polymorphism

- Polymorphism is one thing in many forms.
- It can be achieved by Method Overriding

Polymorphism Implementation



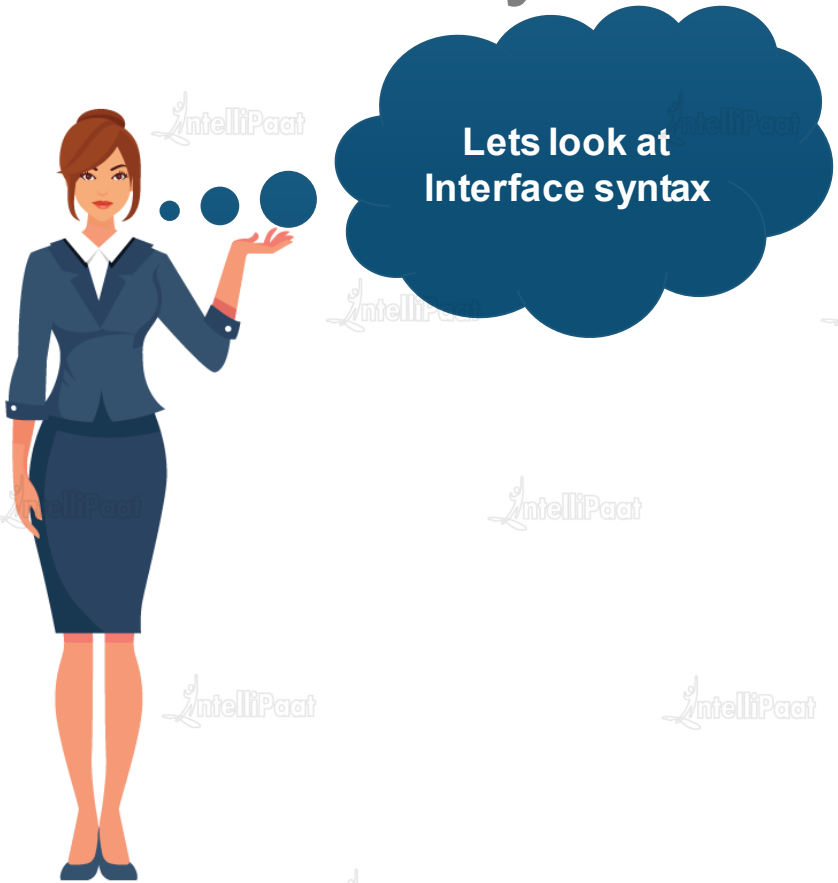
Abstraction

- Data Abstraction is the property by virtue of which only the essential details are displayed to the user.
- Abstraction is achieved by interfaces.

Interface

- Interfaces are declared using the interface keyword and may only contain method signature and varibale declarations.
- An interface is an abstract type that is used to specify a behavior that classes must implement.
- They are similar to protocols.
- Interfaces cannot be instantiated, but rather are implemented

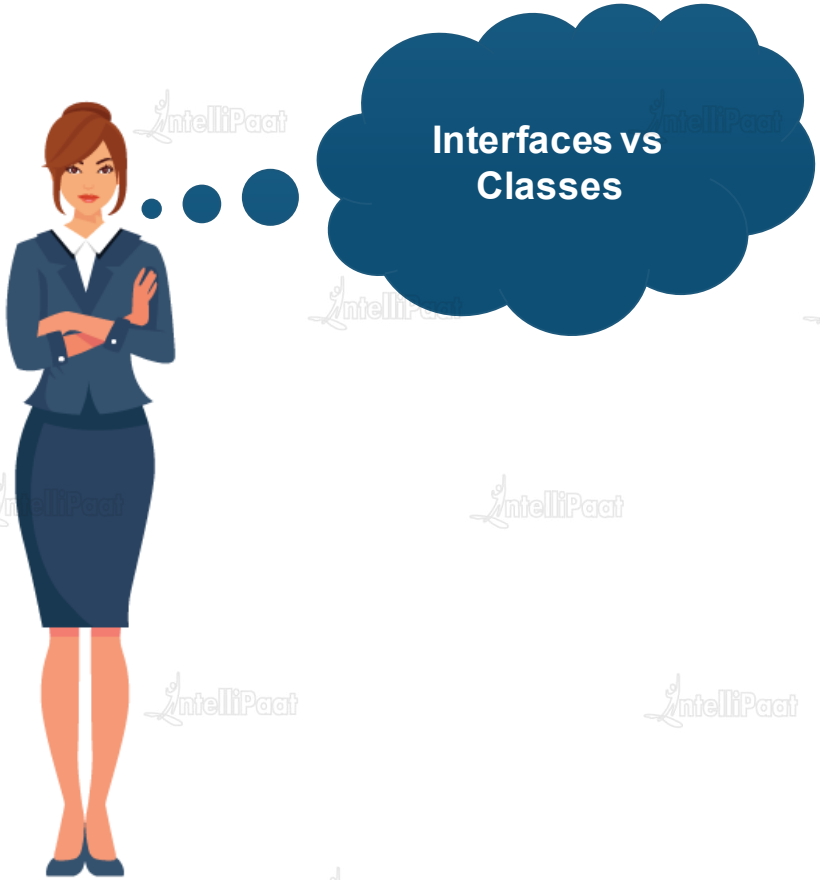
Interface Syntax




```
interface persons{
    fname: string,
    lname: string,
    sayHi: ()=>string
}
```

Usage of Interface in class

Interface vs class



Class

- Define a new type
- Properties (with implementation)
- Methods (with implementation)
- Can be instantiated

Interface

- Define a new type
- Properties (signatures)
- Methods (signatures)
- Can not be instantiated

Encapsulation vs Abstraction

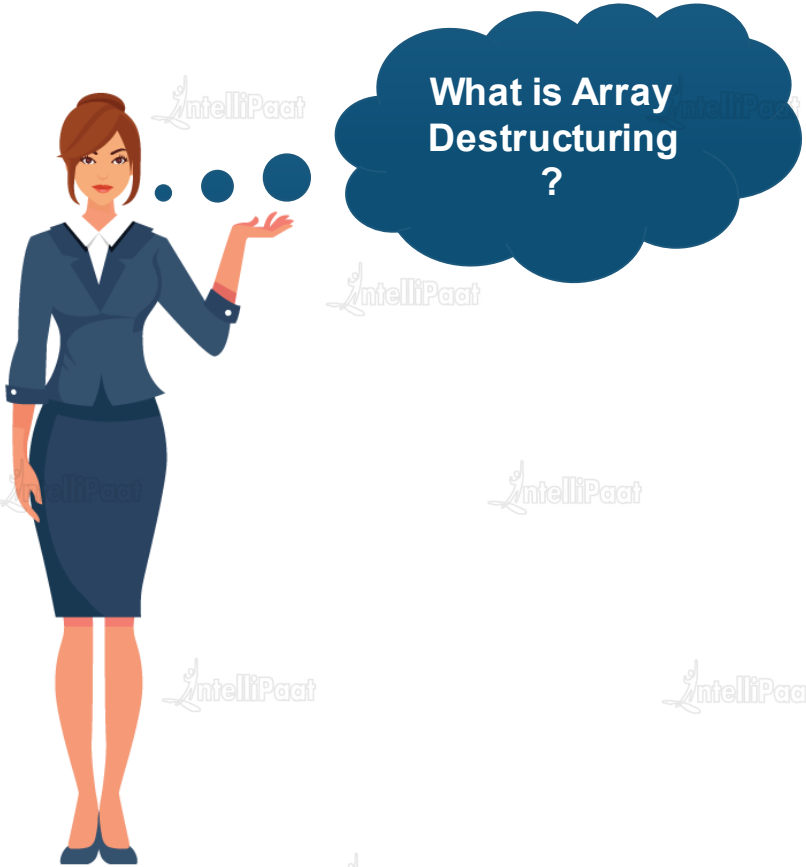
- Encapsulation is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
- While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

Destructuring

Destructuring

The Destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Destructuring



Example:

```
var a, b;
[a, b] = [10, 20];
console.log(a);// expected output: 10
console.log(b);// expected output: 20
```

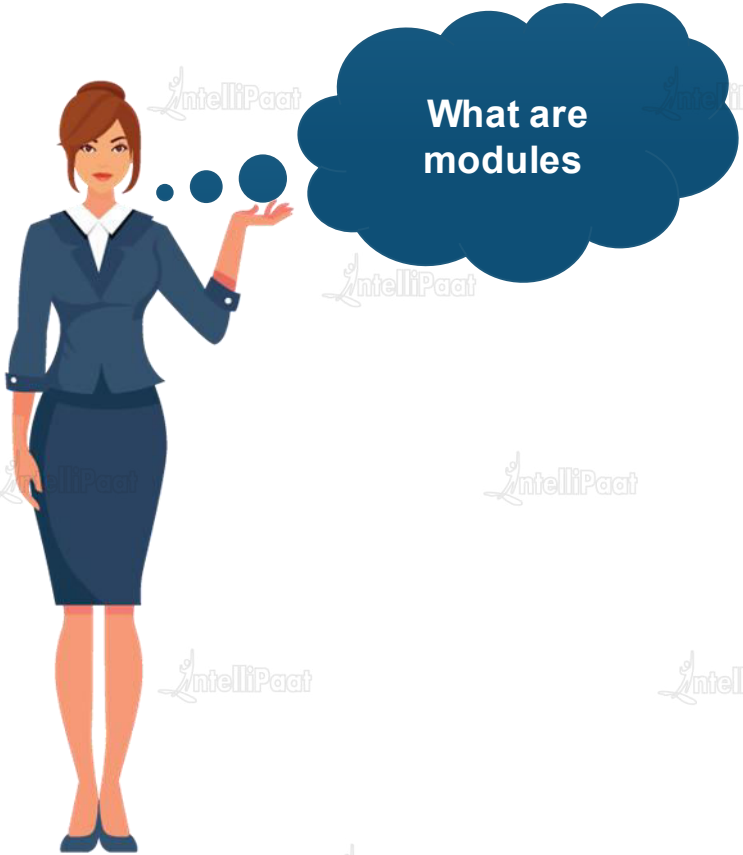
Destructuring

Example:

```
var o = {p: 42, q: true};
var {p, q} = o;
console.log(p); // 42
console.log(q); // true
```

Import and Export

Modules



- Each module is a piece of JS code which contains variable declarations, function declarations, class declarations etc. and by default, these declarations stay local to the module.
- You can mark some of them as export, then other modules can import them.

Export and Import

- The export statement is used when creating JavaScript modules to export classes, functions, objects or primitive values from the module so they can be used by other programs with the import statement.
- A module can import things from other modules using module specifiers like:
  - Relative paths ('../model/user')
  - Absolute paths ('/lib/js/helpers')
- Modules are singletons. Even if a module is imported multiple times, only a single "instance" of it exists.

Multiple Named Export and Import

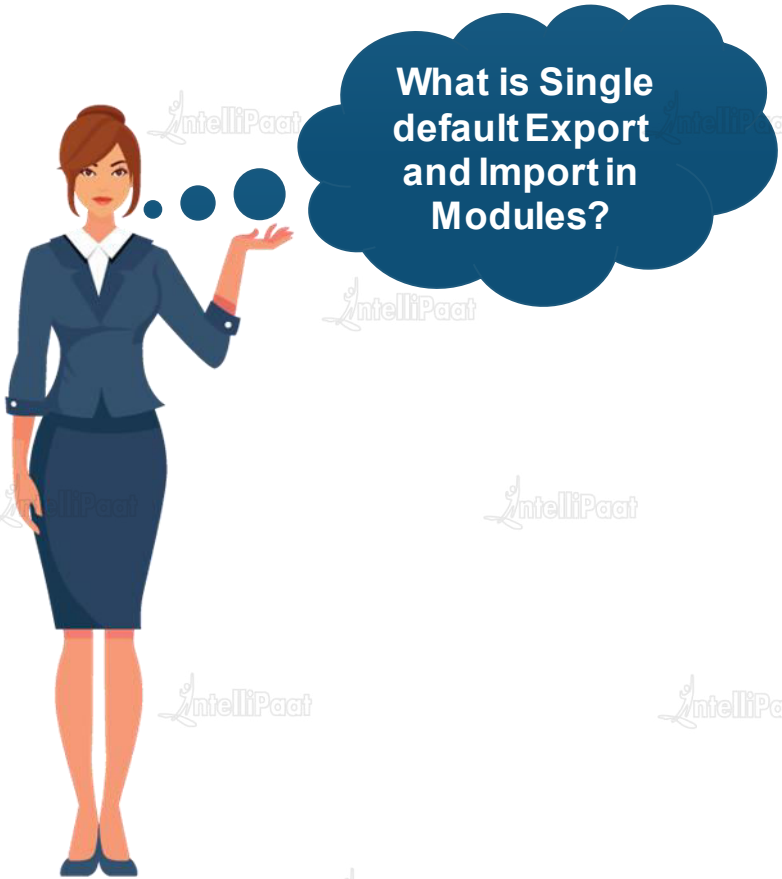
There can be multiple *named exports*:

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x)
{ return x * x; }
export function diag(x, y)
{ return sqrt(square(x) + square(y)); }
export class employees{...}

//------ main.js ------
import { square, diag, employees} from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5

You can also import the complete module:
//------ main.js ------
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

Single default Export and Import

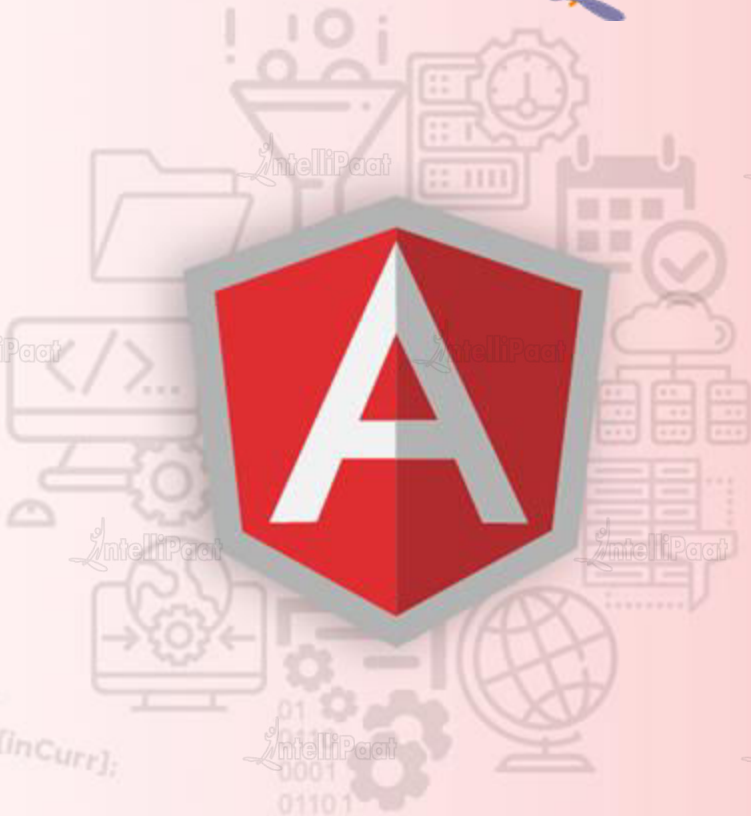



```
//------ myFunc.js ------
 export default function () { ··· } // no semicolon! //
------ main1.js ------
import myFunc from 'myFunc';
myFunc();
Or a class:
//------ MyClass.js ------
export default class { ··· } // no semicolon! //
------ main2.js ------
 import MyClass from 'MyClass';
const inst = new MyClass();
```

Note that there is no semicolon at the end if you default-export a function or a class (which are anonymous declarations).

Map and Set

for,forEach,for in and for of



Looping over an array or an object

We are going to examine how we can loop over arrays, using the methods available to us in the past with ES5 and also the new for-of looping mechanism in ES6.

For and ForEach Method

For one we have the classic for loop, like so:

```
let array = [1,2,3];
for (let i = 0; i < array.length; i++)
{   console.log(array[i]); }
```

We can also use the forEach method on the Array class, like so:

```
let array = [1,2,3];
array.forEach(function (value)
{   console.log(value); });
```

// 1 // 2 // 3

For-in loop

The for-in loop is designed for iterating over an objects properties

```
var obj = {a:1,b:2};
for (let prop in obj)
{  console.log(prop); }
```

// a // b

If we tried to use it with an array, it might initially look like it's working

```
let array = [10,20,30];
for (let index in array)
{   console.log(index); });
```

// 0 // 1 // 2

But if we tried to print out the type of index like so

```
let array = [10,20,30];
for (let index in array)
{   console.log(typeof(index)); };
```

// string // string // string

For-of loop:

Rather than change the way the for-in loops work in ES6 and in the process create a breaking change, instead in ES6 we have a new syntax called for-of

```
let array = [10,20,30];
for (var value of array)
{   console.log(value); }
```

```
// 10 // 20 // 30
```

- This is the most concise way of looping through array elements.
- It avoids all the pitfalls of for–in.
- It works with break, continue, and return

Summarize

- The for–of loop is for looping over the values in an array.
- for–of is not just for arrays. It also works on most array-like objects including the new Set and Map types.

Map and Set

Map

We create a map using the new keyword, like so

```
let map = new
Map();
```

We can then add entries by using the set method, like so

```
let map = new
Map();
map.set("A",1);
map.set("B",2);
map.set("C",3);
```

The set method is also chainable, like so

```
let map = new Map()
.set("A",1) .set("B",2)
.set("C",3);
```

Or we could initialise the Map with a an array of key-value pairs, like so

```
let map = new Map
([ [ "A", 1 ], [ "B", 2
], [ "C", 3 ] ]);
```

Map methods

We can extract a value by using the get method:

```
map.get("A");
// 1
```

We can check to see if a key is present using the has method:

```
map.has("A");
// true
```

We can delete entries using the delete method:

```
map.delete("A");
// true
```

We can check for the size (number of entries) using the size property:

```
map.size
// 2
```

We can empty an entire Map by using the clear method:

```
map.clear()
map.size
// 0
```

Looping over a map

```
let map = new Map([     [ "APPLE", 1 ],     [ "ORANGE", 2 ],     [ "MANGO", 3 ] ]);
```

Using keys():

returns the keys in the map as an array which we can loop over

```
for (let key of map.keys())
{     console.log(key); }
```

```
// APPLE // ORANGE // MANGO
```

Using values():

returns the values in the map as an array which we can loop over

```
for (let value of map.values())
{     console.log(value); }
```

```
// 1: // 2 // 3
```

Using entries():

returns the [key,value] pairs in the map as an array which we can loop

```
for (let e of map.entries()) {
console.log(e[0],e[1]); }
```

```
// "APPLE" 1 // "ORANGE" 2 // "MANGO" 3
```

Set

Sets are a bit like maps but they only store keys not key-value pairs. Sets can only store unique values, so adding a value a second time has no effect.

We create a Set using the new keyword, like so

```
let set = new Set();
```

We can then add entries by using the add method,

```
let set = new Set();
set.add('APPLE');
set.add('ORANGE');
set.add('MANGO');
```

The set method is also chainable, like so:

```
let set = new Set()
.set("A")
.set("B")
.set("C");
```

Or we could initialise the Map with a an array of key-value pairs, like so:

```
let set = new
Set(['APPLE',
'ORANGE',
'MANGO']);
```

Set methods

We can check to see if a value is in a set like so:

```
set.has('APPLE')
```

```
// true
```

We can delete a value from the set:

```
set.delete('APPLE')
```

We can count the number of entries in the set like so:

```
set.size
```

```
// 2
```

We can empty the entire set with the clear method:

```
set.clear();
set.size
```

```
//0
```

Looping over a Set


```
let set = new Set(['APPLE', 'ORANGE', 'MANGO']);
for (let entry of set) {
  console.log(entry); }
```

```
// APPLE // ORANGE // MANGO
```

Nodemon

- Nodemon is a utility that will monitor for any changes in your source and automatically restart your server. Perfect for development.
- Install it using:

  npm install –g nodemon

Demo

India : +91-7847955955

US : 1-800-216-8930 (TOLL FREE)

sales@intellipaat.com

24X7 Chat with our Course Advisor