**Typescript**

TypeScript is a syntactic superset of JavaScript which adds **static typing**.This basically means that TypeScript adds syntax on top of JavaScript, allowing developers to add **types**.

TypeScript being a "Syntactic Superset" means that it shares the same base syntax as JavaScript, but adds something to it.

## **Why should I use TypeScript?**

JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.

In JavaScript, function parameters and variables don't have any information! So developers need to look at documentation, or guess based on the implementation.

TypeScript allows specifying the types of data being passed around within the code, and has the ability to report errors when the types don't match.

For example, TypeScript will report an error when passing a string into a function that expects a number. JavaScript will not.

TypeScript uses compile time type checking. Which means it checks if the specified types match **before** running the code, not **while** running the code.

**How do I use TypeScript?**

A common way to use TypeScript is to use the official TypeScript compiler, which transpiles TypeScript code into JavaScript.

**TypeScript Compiler**

* TypeScript is transpiled into JavaScript using a compiler.
* TypeScript being converted into JavaScript means it runs anywhere that JavaScript runs!

## Installing the Compiler

TypeScript has an official compiler which can be installed through npm.

**--> sudo npm install** **-g** typescript (For Global install)

or

**--> npm install** typescript --save-dev (For Project Use only)

if installed for only Project ?

The compiler is installed in the **node\_modules** directory and can be run with: **npx** tsc.

**--> npx** tsc

**How to create and run typescript file ?**

1. Create an file with extension **.ts**
2. Write some javascript code with typescript addon
3. Run file with --> **tsc** file\_name.ts

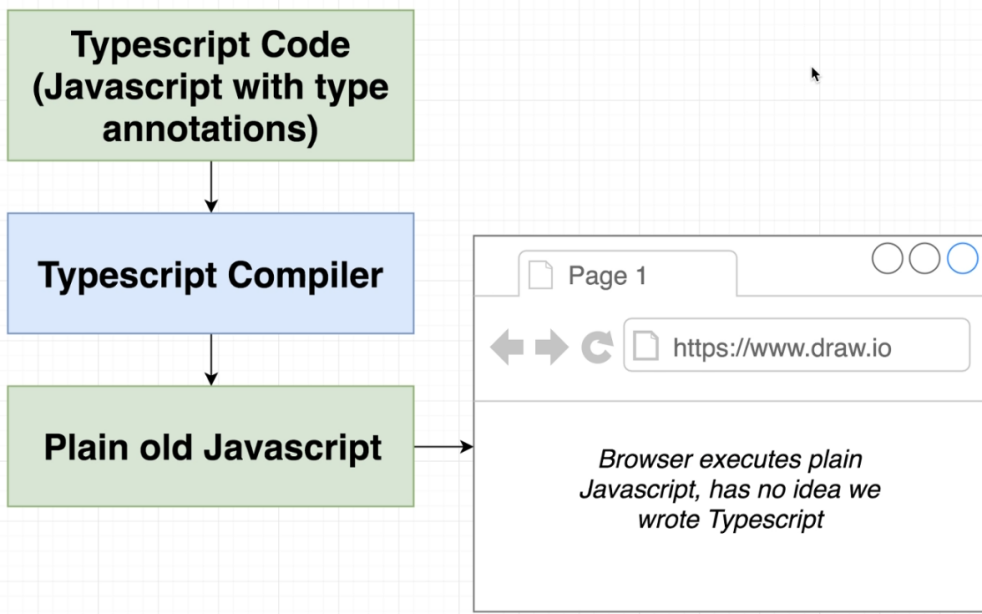
This command will create an javascript file with same file\_name but with extension **.js**

Run file with --> **node** file\_name.js

OR

1. install package ts-node with -->**sudo** **npm install -g** ts-node
2. **ts-node** = **tsc** file\_name.ts **&& node** file\_name.js
3. So simply Run Command **--> ts-node** file\_name.ts

**Working of typescript ?**

****

To check **Live .** Working Convertor

Click Here -> **[Link](https://www.typescriptlang.org/play)**

**Starting TypeScript Example**

**import axios** from “axios”;

**const** url = `https://jsonplaceholder.typicode.com/todos/1`

**What is interface ?**  
interface is an **structure** of varibles with there types

**interface** Todo {

**id**:number;

**title**:string;

**completed**:boolean

}

**What is Todo ?**

Todo is an interface defined on top

it will help typescript to recognise the type of variables

**axios.get**(url).then((response)=>{  
**const** todo = response.data **as** Todo

**const** id = todo.id;

**const** title = todo.title;

**const** completed = todo.completed

**Why types are defined with variables ?**

This is called annotations of variable

annotations are used to help typescript to determine type of variable will be used

**logTodo**(id,title,completed);

})

**const logTodo** = (**id**:**number**,**title**:**string**,**completed**:**boolean**)=>{

**console.log(**`

Id = ${id}

Title = ${title}

Completed Task = ${completed}

`**)**

**Important Note**

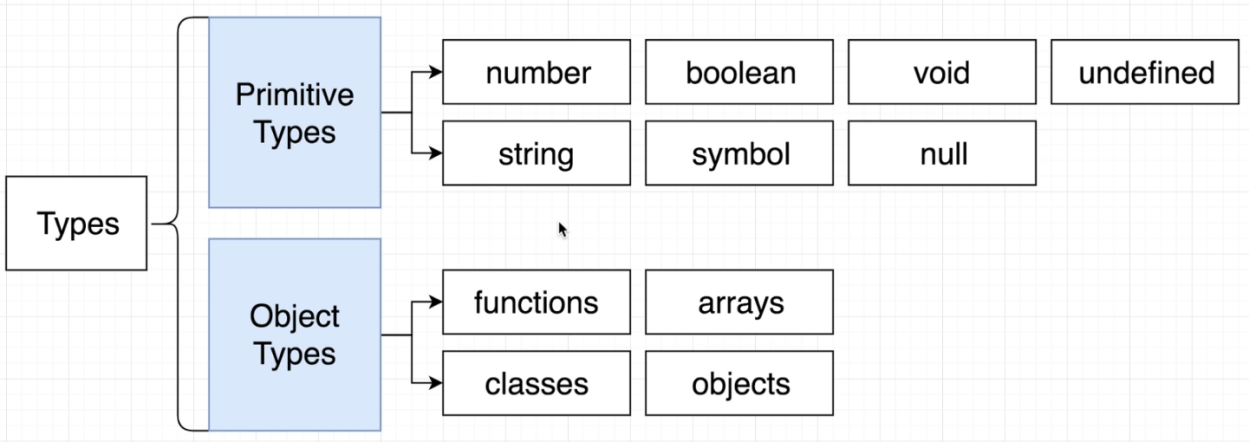
All these steps are for Catching Error by the Typescript and verifying auto type

}

**Understanding Typescript in detail**

**What is Type ?**

Easy way to refer to the different properties + functions that a value has



**Why do we care about types ?**

* Types are used by Typescript compiler to analyze our code for errors
* Types allow other engineers to understand what values are flowing around out codebase

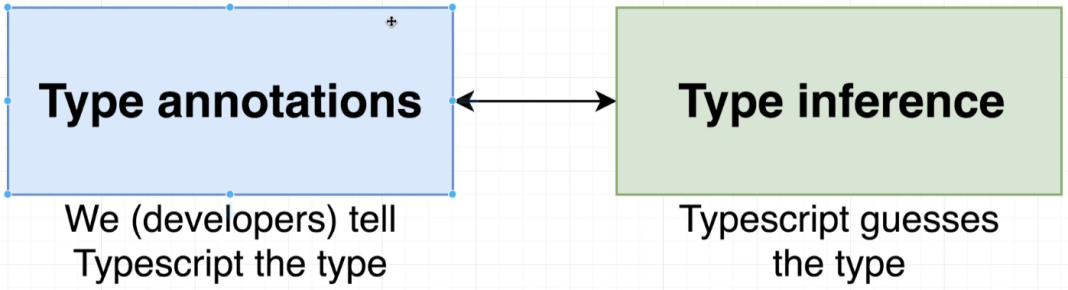
**What** are **Type Annotations** and **Type Inference** ?

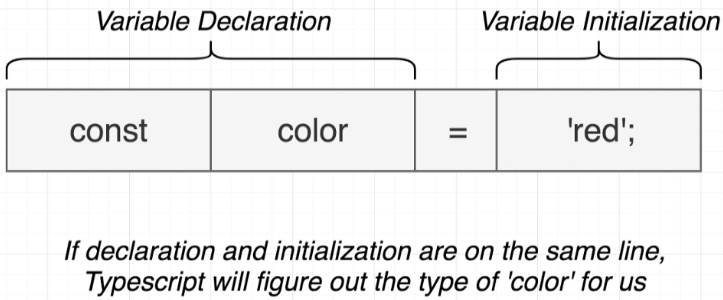
**Type Annotations** (Manual Declare)

Code we add to tell Typescript what type of value a variable will refer to

**Type inference** (Auto Detect)

Typescript tries to figure out what type of value a variable refers to.





**Code Example (Manual Annotations) Used for Complex Structuring of Types**

let cars: **number** = 4;

let carColor: **string** = "Green";

let isAutomatic: **boolean** = true;

let nothingMuch: **null** = null;

let nothing: **undefined** = undefined;

// Combining Different Types with |

let carsDetails: **number** | **string** | **boolean** = 0;

carsDetails = 23;

carsDetails = "Maruti";

carsDetails = true;

// Build in Objects {}

let now: **Date** = new Date();

// Arrays Declaration

const colors: **string[]** = ["Red", "Orange", "Black"];

const myNumbers: **number[]** = [1, 23, 4, 33];

const truths: **boolean[]** = [true, false, true, false];

//Combining Arrays with different Types

const combinedArray: **(string | boolean | number)[]** = ["String", 123, true];

// Classes -> Note Classes will be declared with first capital letter

class Car {}

let car1: Car = new Car();

// Object literals

let points: **{ x: number; y: number }** = {

x: 10,

y: 30,

};

// Functions

const carNumber: **(i: number) => void** = (i: **number**) => {

console.log(i);

};

**Same Code With Inference (Auto Detect Types by Typescript )**

Inference Work if Developer Initialize the Variable with Type of Expected Content

**let** cars = 4;

**Inference (by Initialization)**

Auto Detected Types of

**cars**, **carColor**, **isAutomatic**, **nothingMuch**, nothing

**let** carColor = "Green";

**let** isAutomatic = true;

**let** nothingMuch = null;

**let** nothing = undefined;

// Combining Different Types with |

**Cannot do Inference**

**Initializing is not Posible**

Types Vary while reDefining again and again of different type

**Typescript can’t Determine Types**

**Solution: Annotation Used**

**let** carsDetails: number | string | boolean = 0;

carsDetails = 23;

carsDetails = "Maruti";

carsDetails = true;

// Build in Objects {}

**let** now = new Date();

// Arrays Declaration

**Inference (by Initialization)**

Auto Detected Types of

**colors**, **myNumber**, **truths**

**const** colors = ["Red", "Orange", "Black"];

**const** myNumbers = [1, 23, 4, 33];

**const** truths = [true, false, true, false];

**Inference (by Initialization)**

Auto Detected Type of

**combinedArray**

//Combining Arrays with different Types

**const** combinedArray = ["String", 123, true];

// Classes -> Note Classes will be declared with first capital letter

**Inference (by Initialization)**

Auto Detected Type of

**Car**

**class** Car {}

**let** car1 = new Car();

// Object literals

**Inference (by Initialization)**

Auto Detected Type of

**points** with value type because of initialization {x:number, y:number}

**let** points = {

x: 10,

y: 30,

};

// Functions

**Inference (by Initialization)**

Auto Detected Type of

**carNumber** because of initialization

( i : **number** ) : **void**

**const** carNumber = (i: number) => {

console.log(i);

};

**Type Inference**

1. Typescript guesses the type
2. When to Use
3. Always Try to use

**Type Annotations**

1. We Tell Typescript the type
2. When to Use
3. When we declare the variable in one line and initialize it later
4. When we want a variable to have a type that can’t be inferred
5. When function returns the ‘any’ type and we need to clarify the value

**What is ‘any’ type in TS ?**

Lets Understand it with an example

lets have an --> const **jsonVar** = **‘** **{ “**x**”**: 10, **“**y**”**: 20 **}** **’** ; // It is an String Write Now

lets Conver to json -->const coordinates = JSON.parse(**jsonVar**) result => {x:10,y:20}

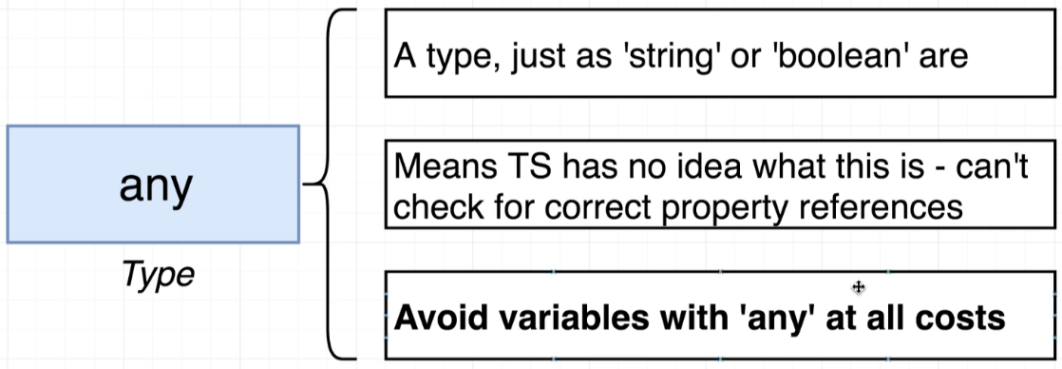
now this result is not defined any where and cannot be defined because TypeScript will never going to run JSON.parse() by its own, for its type of literal it will return.

1. Eg -> **JSON.parse( “**10**” )**  **-->** **10** **(**type **number)**
2. Eg -> **JSON.parse( '**[1,2,3,4,5,2]**' ) --> [ 1,2,3,4,5,2 ] (**type **Array)**
3. Eg -> **JSON.parse( '** { "**x**" :10, "**y**" :"Davi" } **' ) --> { x:10, y: “Davi” } (**type **Object)**

**CONCLUSION**

**No one can predict in advance the type of literal returned by JSON.parse(“”)**

**So ‘any’ type comes from here , TypeScript will predict ‘any’ as an type if this type of situation occours**

****

**Functions in TypeScript**

In TypeScript functions need to be declared with there return type .

**Inference** work only with return type not with the arguments passed in function .

Always try to give type annotations for arguments as well as function return type.

Eg (With Annotations)

**1st way to declare function**

**function addition**(a: number, b: number): number {

return a + b;

}

**2nd way to declare annoymous function**

**const addNum = function** (a: **number**, b: **number**):**number** {

return a + b;

}

**3rd way to declare function**

**let substrac**t: (a: number, b: number) => number = (a: number, b: number) => {

return a - b;

};

Eg (Without Annotation , Just Inference)

Here Inference will work for just return type prediction

**1st way to declare function**

**let add** = (a: **number**, b: **number**) => {

return a + b;

};

**2nd way to declare annoymous function**

**const subNum = function** (a: **number**, b: **number**) {

return a + b;

}

**3rd way to declare function**

**function sub**(a: **number**, b: **number**) {

return a + b;

}

**What is never Type in TS ?**

never Type is used when there is no end of function or the function which is going to only pass to other function

Eg->

**function causeError**(): never {

throw new Error("Not Working");

}

How to do Destructuring with Annotations ?

// destructuring with annotations

**let forcast** = {

date: new Date(),

weather: "Sunny",

};

**function logWeather**(**forcast**: { date: Date; weather: string }): void {

console.log(`

Date = ${forcast.date}

weather is going to be ${forcast.weather}

`);

}

**logWeather(forcast);**

//destructuring

**function logWeatherDesc**(**{ date, weather }**: { date: Date; weather: string }) {

console.log(`

Date = ${date}

weather is going to be ${weather}

`);

}

**logWeatherDesc(forcast);**

**Objects**

Objects can be declared Normally and also can be destructured, **Eg**

**const profile** = {

**firstName:** "Davinder",

**age:** 25,

Annotation Way

only do it if dealing with **any** type

**cordinates:** {

**lat:** 12,

**log:** 20,

},

setAge(**age: number**) {

this**.age** = age;

},

};

// const **{ firstName, age }**: **{ age: number; firstName: string }** = profile;

// const **{ cordinates }**: **{ cordinates: { lat: number; log: number }** **}** = profile;

**Inference Way (Prefered)**

Good Practice for Object Destructuring

**const { firstName, age }** = profile;

**const { cordinates }** = profile;

**console.log**(`

FirstName = ${firstName}

Age of Employee = ${age}

Living Cordinates = { lat:${cordinates.lat} log:${cordinates.log} }

`);

**Arrays**

Arrays in TS is Similar to JavaScript but with an additional feature of Inference the data type of element of array.

Type Guessing Help TS to provide functionality options while hovering over or maping elements of array

Types Annotations in Array

const **carMaker** = [‘ford’,’toyota’,’chevy’]

const **dates** = [new Date(), new Date()]

// 2D Array

const **carsByMake** = **string [] []** = [[“Naming,Conventions”],[“DavinderKumar”,”Deep”],[“Second”,”First”]]

// Help with inference when extracting values

const **car** = carMakers[0] ; ---> will detect the type of element and suggest methods and properties

const **myCar** = carMakers.pop()

// Prevent incompatible values to get assigned

carMakers.**push**(234) ---> Error Detected while assigning wrong type literal

// Help with ‘map’

carMakers.**map**( (**car**:**string**) : **string** => {

**return car**.toUpperCase(); ---> will detect the type of element and suggest methods and properties

})

// Flexible Types

**const importantDates**:**( Date | string | boolean)[]** = **[**new Date()**,** “My Current Date”**,** 233445**]**

importantDates.push(new Date())

importantDates.push(“WorkingFine”)

importantDates.push(true)

**Tuple**

* Tuple is an Array.
* Tuple size if fixed accourding to user as per requirement.
* Every Element in Tuple represent some property or meaning.
* User Must know exact location of element in Tuple for element meaning

EG --> [“Watch”, 200000, India, 5 ]

Here

0 Position = Type of Product

1 Position = Amount of Product

2 Position = Location of buying

3 Position = Rating of Product out of 5

**Why we use Tuple ?**

Tuple is used for storing data in form of rows in database where every column have well defined meaning.

**How to give ‘type alias’ in Tuple ?**

* Alias in tuples in the type used to verify every column type.
* Here Drink is an alias which represent the array of types of element in order
* Here **[‘Sprite’, 200, true]** is an tuple
* Which Represents **[**Name\_of\_drink**,** Sugar\_In\_Drink**,** Contains\_Carbon**]**

**Drink** = [‘string’,number,boolean] **//Type alias**

**const** sprite:**Drink** = [‘Sprite’, 200, true]

**const** coke:**Drink** = [‘CocaCola’, 300, true]

**const** lemon:**Drink** = [‘Lemon Water’, 100, false]

**Interfaces and Classes**

Interface and classes helps Developer for writing strong reusable code in TS

**Define Interface**

Creates a new type or an **Structure**, describing the property names and value types of an object

const **oldCivic** = {

**name:** "civic",

**year:** new Date(),

**Without Interface Structure**

we need to write whole bunch of code for types of varible and every time

**broken:** true,

**summary ():** string {

return `

Name of Vehicle ${this.name}

Year of Manufacture ${this.year}

is it Broken? ${this.broken} `;

},

};

Without **Interface**

const **printVehicle** = ( **vehicle:** {**name:** string;**year:** Date;**broken:** boolean;**summary():** string;}): **void** => {

**console.log**(`Name of Vehicle = ${**vehicle**.name}`);

**console.log**( **vehicle**.summary() );

};

**printVehicle**(oldCivic);

With **Interface**

**interface** Reportable{

**Interface** used for code efficiency

**name:** string;

**year:** Date,

**broken:** boolean,

**summary():** string

}

const **printVehicle** = ( **vehicle**: Reportable):void =>

{

**console.log**(`Name of Vehicle = ${**vehicle**.name}`);

**console.log**( **vehicle**.summary() );

};

Important Note

It is not compelsory that vehicle object must always have all element equal to the elements declare in interface Reportable, it Require minimum 1 element to match which must be within boundries of object vehicle and Interface Reportable

**Code Reusable** using **Interface**

const **Drink** = {

**color**: “Brown”,

**carbonated**: true,

**sugar:** 50

**summary () :** **string** {

**return** `My Drink has ${this.sugar} amount of sugar`

}

}

// **Reportable Interface** and Drink Object has summary function Common, So we can reuse it

const **printDrink**= ( **drink**: Reportable):void =>

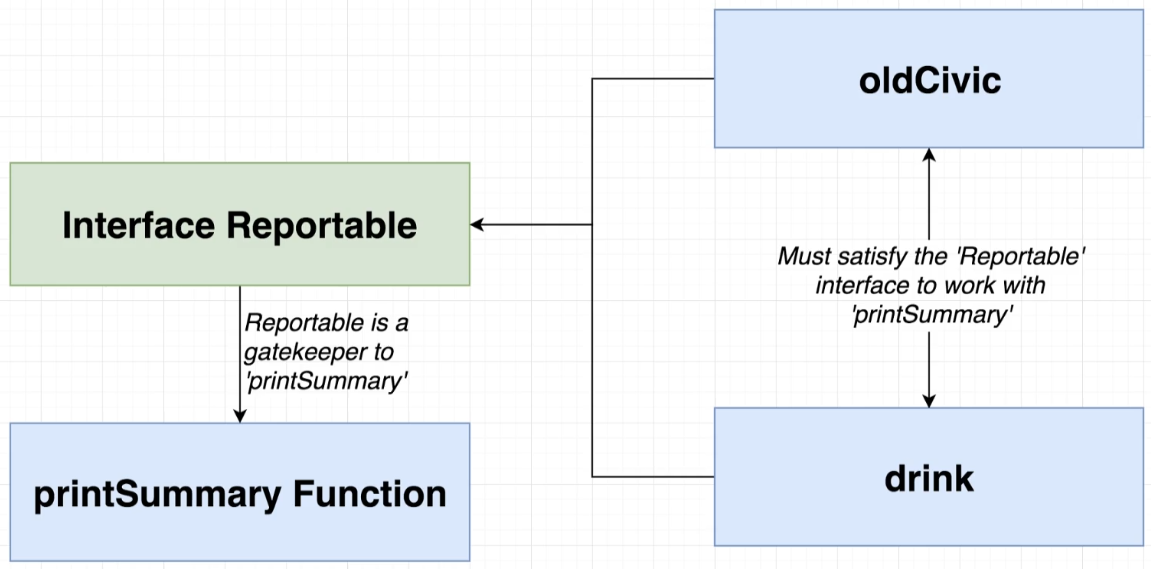
{

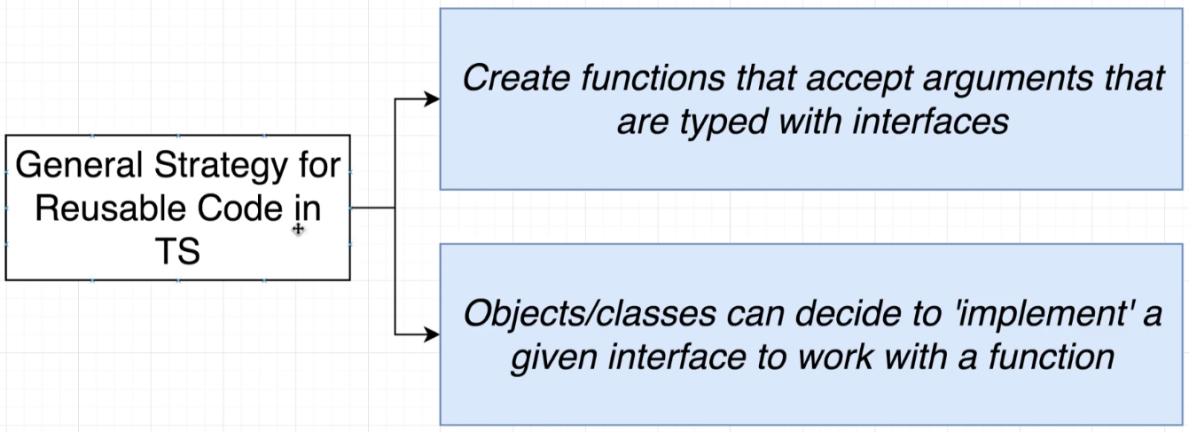
**console.log**( **drink**.summary() );

};

printDrink(Drink);

**Diagram Explanation**





**Classes**

**Class** is an Blueprint to create a object with some feilds (values) and methods (functions) to represent a ‘thing’

**constructor() Read This [link](https://medium.com/@eamonocallaghan/prototype-vs-proto-vs-prototype-in-javascript-6758cadcbae8)**

* **constructor** in class is used to initialize the properties or run the methods located inside the class, Remember Constructor is the first thing which automaticaly runs when we create an new Object with **‘new’** Keyword so contructor is like an indication to the memory to provide the enough space with respect the defined properties and methods in class instance to the **OBJECT** created with **new** Keyword
* A class cannot have more than one constructor method.

**‘new’ Keyword**

* It will help to call **constructor** of **class**
* It is an way the **object** will get connected to the class **methods** and **properties**.
* Object will attain its own memory scope.
* Class will help object to get its type.
* use **new** keyword in front of class before calling it

**super()**

* super method is located inside constructor of derived class (Child Class)
* It is used to call constructor of extended class (Parent Class) from child class
* It is very Usefull if Parent Class construtor require initialization arguments so it can be passed through super method located at child class constructor.
* While we create object of Child class first step is to go to the constructor of child class and call super() first and proceed the following steps in constructor of child class

**class CarDetails**{

**type**= 'OK';

**ammount** = 0;

**constructor**(type,ammount){

**console.log**("Worked Parent")

this.type = type;

this.ammount = ammount;

}

}

**class Car1** extends **CarDetails**{

**constructor**(type,ammount){

**console**.log("Child Before Super")

**super**(type,ammount)

**console.log**("Child After Super")

}

**printCarDetails**=()=>{

**console.log**(this.type)

}

}

Result

->Child Before Super

->Worked Parent

->Child After Super

->BMW

**const** car = **new** Car1("BMW",20000)

**car**.printCarDetails()

**Eg**

**Class Modifiers**

* **public :** This method can be called any where any time
* **private :** This method can only be called by other methods in this class
* **protected :** This method can be called by other methods in this class, or by other methods in class classes

**Example** = **Class Modifiers + Types**

**class** Vehicle {

**constructor**(**public** color: string) { }

**protected** honk(): void {

console.log('beep');

}

}

const **vehicle** = new Vehicle('orange');

**console.log**(vehicle.color);

**class** Car extends Vehicle {

**constructor**(**public** wheels: number, color: string)

{

**super**(color);

}

**private** drive(): void {

**console**.log('vroom');

}

**startDrivingProcess**(): void {

**this**.drive();

**this**.honk();

}

}

**const** car = new Car(4, 'red');

**car**.startDrivingProcess();

**It is an Simple way of Writing**

**color** : string

**constructor** ( color : string ){

**this**.color = color

}

**It is an Simple way of Writing**

**wheels** : string

**color**: string

**constructor** ( **color** : string, **wheels**: string )

{

**this**.color = color

**this**.wheels = wheels

}

**Lets WireUp the TypeScript**

**Very Important**

If You are using **TypeScript** with **React Projects**

Use **Typescript** friendly package

**Eg**

**npm install** axios

**npm install** @types/axios --> Specially for TypeScript

**npm install** faker

**npm install** @types/faker --> Specially for TypeScript

**npm install** @types/google.maps --> Specially for TypeScript

**What are Type Defination Files ?**

These are files which help typescript to detect types of every methods,properties,classes of using library

To Search -> Types Defination Files go to

<https://www.npmjs.com/>

Search with --> @types/**Package\_name**

**These @types Packages will help Typescript to get Defined Types to every packages functions and properties**