**Typescript**

**Terminal**

C:\ > **npm install -g typescript** > Typescript son sürüm global kurulum

C:\ > **npm install -g** [**typescript@4.4.2**](mailto:typescript@4.4.2) > Typescript belli bir sürüm global kurulum

C:\ > **npm uninstall -g typescript** > Typescript’i sistemden kaldırmak

C:\ > **tsc -v** > Typescript versiyon kontrol

C:\ > **tsc main.ts** > TS kodlarını JS kodlarına dönüştürür.

**Genel**

🡪 JS’den daha güzel/modern bir söz dizimi sunar. (Superset of Javascript)

🡪 Derleme aşamasında anlık kod hatalarını bildirir. (Compile-time Error)

🡪 JS’de bir değişkene sonradan farklı bir değer atanabilir ama TS buna izin vermez.

**Değişkenler**

let a: number = 5;

let b: string = 'a';

let c: boolean = true;

let d: any;

let e: number[] = [1,2,3];

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

let g: any[] = [1,'a',true];

let h: [string,number,boolean] = ['a', 5 , false]; // tuple

🡪 Değişken ile birlikte tür ve değer de atanabilir.

🡪 e değişkenininde diziye number türü dışında eleman eklenemez.

🡪 e ve f değişkenleri yazımları farklı olsa da benzer özellik taşır.

🡪 **tuple** olarak da bilinen h değişkeninde tür ve değer sıralaması aynı olmalıdır.

**Enum**

const krediPayment = 0;

const havalePayment = 1;

const eftPayment = 2;

// Enum

enum Payment {kredi, havale, eft};

let kredi = Payment.kredi;

let havale = Payment.havale;

let eft = Payment.eft;

🡪 Enum’da değer atanmadığı takdirde index değerleri çağırılır. Ancak **enum Payment {kredi=0, havale=1, eft=5}** şeklinde değer de atanabilir.

**Tür**

let message = 'Hello World';

message.length;

🡪 Değişken tanımlanırken değer atandığından **string** bir ifade olduğu tespit edilebildi ve **'message.'** yazıldığında string metodları görüntülendi.

let message;

message = 'Hello World';

let length = (<string>message).length;

let length = (message as string).length;

🡪 Değer ataması sonradan yapıldığından türü **'any'** kabul edildi ve string metodları görüntülenmedi. Bunun için tür dönüşümü yapılmalıdır.

**Fonksiyonlar**

function getAverage(a, b, c) {

    const result = (a + b + c) / 3;

    return `result : ${result}`

}

getAverage(10, 20, 30);

🡪 **getAverage()** fonksiyona sayısal olmayan bir değer yollandığında kod çalışırken hata verecektir.

function getAverage(a: number, b: number, c:number): string {

    const result = (a + b + c) / 3;

    return `result : ${result}`

}

getAverage(10, 20, 30);

🡪 Ancak kod çalışmadan önce geliştirme aşamasında hataları görmek ve önlem almak için değer atanabilir.

🡪 Aşağıda hem sayısal bir değer dışında veri girişi olursa hata vermesini, hem de fonksiyonun **string** bir sonuç döndürmesini sağladık.

function getAverage(a: number, b: number, c?:number): string {

    const result = (a + b + c) / 3;

    return `result : ${result}`

}

getAverage(10, 20);

🡪 Normalde fonsiyona 3 değer girilmesi zaruriydi. Ancak üçüncü değeri **c?** yaparakopsiyonelhale getirdik.

function getAverage(a: number, b: number, c?: number): string {

    let total = a + b;

    let count = 2;      // parametre sayısı

    if (typeof c !== 'undefined') {

        total += c;

        count++;

    }

    const result = total/count;

    return `result : ${result}`

}

getAverage(10, 20, 30);

🡪 Eğer **c** parametresinin typeof’u **undefined** değilse demek ki bir değer girilmiş demektir. O halde total’e c de eklensin ve count (parametre sayısı) +1 artsın.

***Rest Parameters***

const getAverage = (...a: number[]): string => {

    let total = 0;

    let count = 0;

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

        total += a[i];

        count++;

    }

    const result = total / count;

    return `result : ${result}`

}

getAverage(10, 20, 30, 40);

🡪 Parametre bilgisi belli değilse, gelecek olan tüm parametreler **rest parameter** ile hazırlanarak fonksiyon içinde hesaplamaya tabi tutulabilir.

🡪 Yukarıda **a** parametresini sayı dizisi olarak belirledik.

🡪 Başlangıçta toplam ve parametre sayısı 0’dır.

**Interface**

let travelTo = (point: {}) => {

    //..

}

let getDistance = (pointA: { x: number, y: number}, pointB: { x: number, y: number}) => {

}

travelTo({

    x: 1,

    y: 2

});

🡪 Verilen koordinatlara göre istediğimiz noktaya gitmemizi sağlayan bir metod varsayalım.

interface Point {

    x: number;

    y: number;

}

let travelTo = (point: Point) => {

    //..

}

let getDistance = (pointA: Point, pointB: Point) => {

    //..

}

travelTo({

    x: 1,

    y: 2

});

🡪 Her bir **point** için parametreleri teker teker girmek yerine bir **interface** tanımlanabilir.

interface Point {

    x: number;

    y: number;

}

interface Passenger {

    name: string;

    phone: string;

}

interface Vehicle {

    currentLocation: Point;

    travelTo(point: Point): void;

    getDistance(pointA: Point, pointB: Point): number;

    addPassenger(passenger: Passenger): void;

    removePassenger(passenger: Passenger): void;}

🡪 Bir fonksiyon interface içinde belirtilirse **metod** olarak tanımlanır. Gövdesi yer almaz, sadece şeması belirtilir.

🡪 Herhangi bir değer döndürmeyen işlevler için **'void'** type kullanılır.

**Class**

🡪 Interface örneğindeki TS kodlarını JS kodlarına dönüştürdüğümüzde bir çıktı alamayız, bunun için class tanımlamaları yapılması gerekir.

interface Point {

    x: number;

    y: number;

}

interface Vehicle {

    currentLocation: Point;

    travelTo(point: Point): void;

}

class Taxi implements Vehicle {

    //..

    currentLocation: Point;

    travelTo(point: Point): void {

        console.log(`Taksi X: ${point.x} konumundan Y: ${point.y} konumuna gidiyor.`)

    }

}

class Buss implements Vehicle {

    //..

    currentLocation: Point;

    travelTo(point: Point): void {

        console.log(`Otobüs X: ${point.x} konumundan Y: ${point.y} konumuna gidiyor.`)

    }

}

🡪 **Taxi** ve **Bus** araçlarını tanımladığımız class’lar olduğunu varsayalım. Bu class’larda yer alan bazı temel tanımları **implements** kullanarak **interface Vehicle**’dan aktarabiliriz.

**Object**