TypeScript İlgili Not
Öğrendiğimiz şeylerin üzerinden geçmek gerekirse.

## Tipler:

- Boolean
  Mantıksal değerler. true ya da false

- Number
  Her türlü sayısal değer. Farklı tipte sayısal değerler için(decimal, binary) kullanım aşağıdaki gibidir.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
```

- String
  Metin ifadeler. Örneğin: "Falanlar filanlar"

- Array
  Diziler. Tanımlanma şekline göre her şeyin dizisi olabilir. `[1,2,3]` ya da `["1", "elma", [3,5], {deneme:'metin'}]`
- Tuple
  Eleman sayısı sınırlı dizi tanımlaması diyebiliriz şimdilik.

- Enum
  Detaylı bir şekilde değineceğiz ileride

- Object
  Klasik JavaScript objesi, tabi bunu da **interface** yapısı sayesinde belirli bir tip yapısına dönüştürüp özelleştirebiliyoruz.
- Unknown
- Any
  Her türlü veriyi atayabileceğimizi ifade eder. Tip bilgisi any olan değişkenlere tip sınırlaması olmadan her şeyi atayabiliriz.

- Void
- Null and Undefined
- Never
  Peki değişkenler tip bilgisi ile nasıl tanımlanıyor. Burada iki yöntem var. İlki değişkene verdiğimiz ilk değer ile değişkenin tipini belirlemiş oluyoruz. Örneğin;

```
let degisken = 123;
```
burada değişkene sayısal bir değer atayarak değişken'i sayısal bir tip olarak belirledik.

Diğer yöntem ise  değer atamadan önce belirlemek. Bunun için ise değişken adını belirttikten sonra ":" yazıp devamında tip bilgisini ekliyoruz. Örneğin:

```
let degisken2: string= "merhabalar"
```
Bu yöntem ile dizi tanımlarken de dizi elemanının tip bilgisini belirttikten sonra "[]" parentezleri ile dizi olduğu belirtilir.

```
let dizi : number[] = [1,2,5,9]
```

**Interface**

Bu standart tiplerin yanı sıra en çok faydalandığımız tip ise javascript objeleri. Obje tanımlarında tip bilgisini ilk yöntemdeki gibi ilk değer bilgisini vererek yapabilsek de her durumda bunu kullanamayız. Bir objeyi özelleştirmek istediğimizde interface yapısını kullanırız. Bir kişi objemizin sayısal bir yaş bilgisini ve string bir isim bilgisini tutmasını isteyelim. Öncelikle bu yapıda bir interface tanımı yaparız

```
interface IKisi{
    ad:string;
    yas:number
}
```

daha sonra objemizin tip bilgisine bu interface'i belirtiriz.

```
let kisi: IKisi = {ad:... , yas: ...}
```
interface tanımında belirttiğimiz her değişken bu interface ile belirtilmiş her objede olması zorunlu alanlardır. Eğerki bir alanın olması zorunlu olmayacak şekilde belirtmek istersek alan adından sonra "?" işareti eklenir. Bu sayede bu alanın olma zorunluluğu kaldırılmış olur.

**Birden fazla tip belirleme**

Bazı durumlarda değişkenimize birden farklı tipde veri atamak isteyebiliriz. Bu tarz durumlarda tipleri birleştirebiliriz. Örneğin bir değişkene hem sayısal hem de mantıksal değer atayabilmek isteyelim, değişken tanımında tiplerin aralarına "|" işareti ekleyerek bitleştiririz.

```
let bilesikDegisken: number | boolean;
```

Tabi her seferinde böyle uzun uzun yazmak istemeyiz. Bu tipleri birleştirerek kendi tip tanımımızı oluşturabiliriz. Bunun için kullanacağımız ifade ise;

type
örneğin sayısal ve mantıksal tip birleşimini sık sık kullanacağız diyelim. Bunun için "numbool" adında bir tip oluşturabilir ve bu tip tanımını kullanabiliriz.

```
type numbool = number | boolean;
let bilesikDegisken: numbool;
```

## Fonksiyon tanımlama

Tiplerden sadece değişken oluştururken değil fonksiyon tanımlarken de faydalanabiliriz. Böylece Fonksiyonun alacağı parametrelerin tiplerini ve fonksiyonun döndüreceği verinin tipini belirtebiliriz.

Fonksiyonun sadece aldığı parametrelerin tiplerini doğrudan parametre isimlerinin hemen peşi sıra belirtebiliriz.

```
function IlkFonksiyon(param1: number, param2:string, param3: number[])
{
    .....
}
```

Diyelim ki fonksiyonumuz belirli tipte bir veri dönderecek. O zaman da parametre tanımını yazdıktan sonra da ":" işaretini ekleyip dönüş tipini yazarız. Örneğin fonksiyonumuz mantıksal bir ifade döndürecek olsun;

```
function IkinciFonksiyon: (param1: number, param2:string, param3: number[]): boolean
{
    .....
}
```

Ok Fonksiyonlarında değişken adından sonra ":" işareti eklenir parametre tipleri tanımlandıktan sonra "=>" ifadesi eklenir ve dönüş tipi belirtilir.

```
const UcuncuFonksiyon: (param1:number, param2:boolean)=> number = (param1, param2) => {
...
}
```

Interface tanımlamada olduğu gibi bir fonksiyonda bir parametrenin zorunlu olmamasını istediğimiz zaman parametre tanımı yaparken ":" işaretinden önce "?" işareti ekleriz. Bu sayede bu parametre zorunlu olmaktan çıkarılır.

Değinmediğim veri tipleri uygulama boyunca kullanmadığımız tipler. Kafa karışıklığı olmaması için detaylandırmadım. Gelen soru ya da taleplere göre bu kısmı birlikte geliştirebiliriz.