Temel Angular Eğitimi

Umut ÇAKIR

umutcakirbm@gmail.com

Temel Angular eğitiminde neler var?

- Front-end ve Back-end ayrımı
- Javascript ekosistemi ve Scope / Closures / Hoisting
- Variables / Types / Values
- JS Objects & Prototypes
- Angular kurulumu ve temel özellikleri
- Single Page Application (SPA) kavramı
- Angular CLI Kullanımı
- Angular Component Kavramı
- Lifecycle Hook
- Angular Template
- Angular kullanarak CRUD işlemlerinin yapıldığı sayfalar

FE vs BE ?

Front-end ve back-end, web geliştirmesinde iki önemli kavramdır. Front-end, kullanıcının bir web sitesi veya uygulama ile etkileşimde bulunduğu arayüzü oluşturan kısım iken, back-end ise kullanıcının göremediği veri işleme, veri tabanı yönetimi, sunucu tarafı kodlama ve diğer arka planda çalışan işlevleri içeren kısımdır.

- Front-end, kullanıcının bir web sitesi veya uygulamayla etkileşimde bulunduğu arayüzü oluştururken, back-end kullanıcının göremediği veri işleme, sunucu tarafı kodlama ve veritabanı yönetimi gibi işlevleri gerçekleştirir.
- Front-end, HTML, CSS ve JavaScript gibi teknolojilerle çalışırken, back-end daha çeşitli programlama dilleri ve teknolojileri kullanabilir, örneğin Python, Ruby, Java, PHP, .NET gibi.
- Front-end, kullanıcı deneyimini iyileştirmek için tasarım, kullanılabilirlik ve performans optimizasyonlarına odaklanırken, back-end veri işleme performansı, güvenlik önlemleri ve sunucu kaynaklarının verimli kullanımı gibi konulara odaklanır.
- Front-end ve back-end, birbirleriyle iletişim kurarak bir web sitesi veya uygulamanın tam olarak çalışmasını sağlar. Front-end, kullanıcının taleplerini back-end'e ileterek gerekli verilerin alınmasını sağlar, ardından back-end bu verileri işleyerek front-end'e geri gönderir.

Front-end ve Back-end Kavramları

Front-end ve Back-end kavramlarını daha yakından inceleyelim.

Web Geliştirme Nedir?

- **Web Geliştirme:** Web geliştirmesi, web siteleri ve web uygulamalarının oluşturulması, tasarlanması ve kodlanması sürecidir. İnternet üzerindeki içeriğin ve etkileşimin temelini oluşturur.
- **İstemci-Server Modeli:** İnternet üzerindeki iletişim, istemci (client) ve sunucu (server) arasında gerçekleşir. İstemci, kullanıcının cihazıdır ve web sayfasını tarayıcısı üzerinde görüntüler. Sunucu ise, istemciden gelen istekleri işler ve gerekli yanıtları gönderir.
- **HTML (HyperText Markup Language):** HTML, web sayfalarının yapısal olarak nasıl oluşturulduğunu belirlemek için kullanılan standart bir işaret dilidir. Web sayfalarının metin, resim, bağlantı ve diğer içerikleriyle birlikte nasıl organize edildiğini tanımlar.
- **CSS (Cascading Style Sheets):** CSS, web sayfalarının nasıl görüneceğini belirlemek için kullanılan bir stil dilidir. HTML'e uygulanarak, metin biçimlendirme, renkler, düzenler ve görsel efektler gibi özelliklerin kontrolünü sağlar.
- **Javascript:** JavaScript, web sayfalarında etkileşimli öğeler ve dinamik içerikler oluşturmak için kullanılan bir programlama dilidir. Kullanıcı etkileşimini kontrol etmek, form doğrulama, animasyonlar, veri işleme gibi işlevleri gerçekleştirebilir.

Web Geliştirmede Front-end Rolü

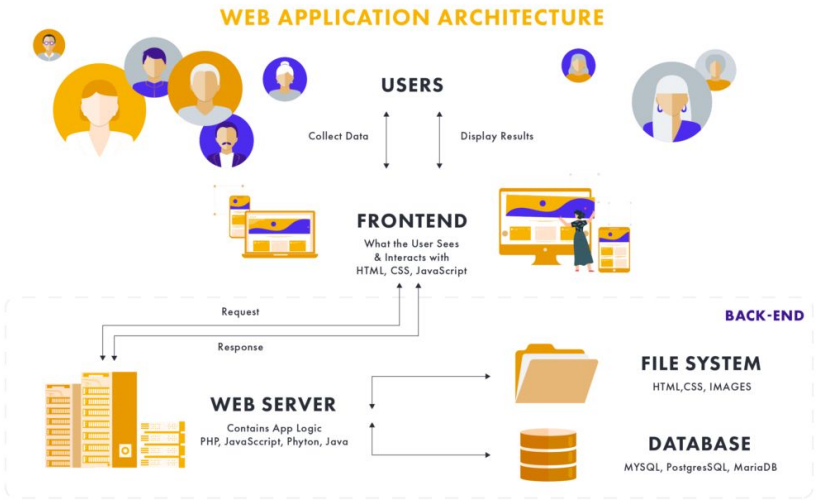
- **Kullanıcı Arayüzü Tasarımı:** Front-end, kullanıcıların bir web sitesi veya uygulama ile etkileşime girdiği arayüzün tasarımını oluşturur. Bu, sayfaların düzeni, renkleri, yazı tipleri, grafikler, düğmeler ve diğer görsel öğelerin yerleşimi gibi unsurları içerir.
- **HTML Yapısının Oluşturulması:** Front-end geliştiricisi, HTML (HyperText Markup Language) kullanarak web sayfalarının yapısal olarak nasıl oluşturulacağını belirler. HTML etiketlerini kullanarak metinleri, başlıkları, paragrafları, bağlantıları ve diğer içerikleri organize eder.
- **CSS ile Stil Uygulaması:** Front-end, CSS (Cascading Style Sheets) kullanarak web sayfalarının nasıl görüneceğini belirler. CSS ile metin biçimlendirme, renkler, düzenler, boyutlar ve görsel efektler gibi stil özellikleri uygulanır.
- **JavaScript ile Etkileşim ve Dinamizm:** Front-end geliştiriciler, JavaScript kullanarak web sayfalarına etkileşim ve dinamizm ekler. JavaScript'i kullanarak animasyonlar, form doğrulama, kullanıcı girdilerini işleme, veri gösterme ve diğer etkileşimli özellikleri sağlar.
- **Tarayıcı Uyumluluğu ve Optimizasyon:** Front-end geliştiricisi, web sayfalarının farklı tarayıcılarda ve cihazlarda tutarlı bir şekilde çalışmasını sağlamak için tarayıcı uyumluluğuna dikkat eder. Ayrıca, sayfaların yüklenme hızını ve performansını optimize etmek için gerekli önlemleri alır.
- **Mobil Uyumlu Tasarım:** Günümüzde mobil cihaz kullanımı arttığından, front-end geliştiriciler mobil uyumlu tasarımlar oluşturur. Sayfaların farklı ekran boyutlarına uyum sağlaması ve mobil cihazlarda iyi bir kullanıcı deneyimi sunması önemlidir.

Web Geliştirmede Back-end Rolü

- **Veri İşleme ve Mantıksal İşlevler:** Back-end geliştiricisi, kullanıcının veri işleme ihtiyaçlarını karşılar. Kullanıcının taleplerini işler, gerektiğinde veritabanından veri çeker, işlemleri gerçekleştirir ve sonuçları kullanıcıya geri gönderir. Örneğin, kullanıcının bir form gönderdiği durumda, back-end bu verileri alıp işler ve sonucunu kullanıcıya gönderir.
- **Sunucu Tarafı Kodlama:** Back-end geliştiricisi, sunucu tarafında çalışan kodları oluşturur. Sunucu tarafı kodlama genellikle daha karmaşık işlemleri gerçekleştirmek için kullanılır ve kullanıcı etkileşimlerini yönetir. Bu kodlama, kullanıcının taleplerine yanıt olarak çalışır ve işlevleri yerine getirir.
- **Veritabanı Yönetimi:** Back-end, veritabanı yönetimi ile ilgilenir. Veritabanı oluşturma, tabloları ve ilişkileri tasarlama, veri ekleme, okuma, güncelleme ve silme işlemleri (CRUD) gibi veritabanı işlemlerini gerçekleştirir. Bu sayede kullanıcı verilerini tutar ve yönetir.
- **API Entegrasyonları:** Back-end geliştiricisi, dış hizmetler ve API'lerle entegrasyon sağlar. Bu entegrasyonlar, kullanıcının taleplerini karşılamak için harici servislere erişimi ve veri alışverişini sağlar. Örneğin, ödeme işlemleri için bir ödeme ağ geçidi API'sini entegre etmek veya haritalama hizmetleri için bir harita API'sini kullanmak gibi.
- **Güvenlik ve Kimlik Doğrulama:** Back-end, güvenlik önlemlerini uygular ve kullanıcı kimlik doğrulama süreçlerini yönetir. Kullanıcı bilgilerini güvende tutmak, yetkilendirme ve kimlik doğrulama işlemlerini gerçekleştirmek, veri güvenliğini sağlamak gibi görevler back-end tarafında yer alır.

Web Geliştirmede İşbirliği

- **Front-end:** Web tasarımcılarıyla yakın işbirliği, kullanıcı arayüzü tasarımı.
- **Back-end:** Veri tabanı yöneticileri, sistem yöneticileri ve diğer geliştiricilerle işbirliği.



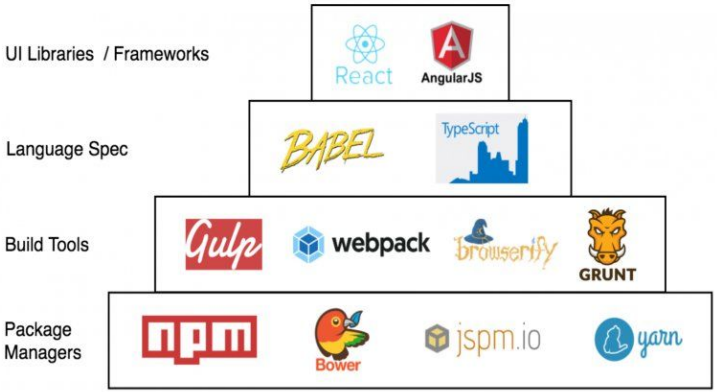
Javascript Ekosistemine Giriş

Javascript ekosisteminde neler bulunur, bunları inceleyeceğiz.

Javascript Ekosistemi

- JavaScript ekosistemine giriş, JavaScript ile çalışan geliştiriciler için mevcut olan çeşitli araçlar, çerçeveler (framework), kütüphaneler ve kaynakların genel bir değerlendirmesini ifade eder. Bu, JavaScript programlama dilinin etrafında oluşturulan tüm ekosistemi kapsar ve hem front-end hem de back-end geliştirmeyi içerir.

- JavaScript ekosistemi son derece çeşitlidir ve sürekli olarak gelişmektedir. Geliştiricilere web uygulamaları, mobil uygulamalar, sunucu tarafı uygulamalar ve hatta masaüstü uygulamaları oluşturmak için geniş bir seçenek yelpazesi sunar.



Javascript Ekosistemi

- **Javascript Dili:** Ekosistemin çekirdeğini JavaScript programlama dili oluşturur. JavaScript, web tarayıcılarında (istemci tarafında) ve Node.js ile sunucu tarafında çalışan çok yönlü, dinamik bir dilidir. Geliştiricilere web sayfalarına etkileşim ve dinamik işlevsellik eklemelerine olanak tanır.

- **Çerçeveler (Framework) ve Kütüphaneler:** JavaScript ekosistemi, geliştiricilere geliştirmeyi kolaylaştıran önceden oluşturulmuş işlevsellikler ve araçlar sunan çeşitli çerçeveler ve kütüphanelerle doludur. React, Angular ve Vue.js gibi çerçeveler, sağlam front-end uygulamaları oluşturmak için popüler seçeneklerdir. Node.js, sunucu tarafı JavaScript geliştirmesini mümkün kılar. Ayrıca, veri manipülasyonu, grafik oluşturma, animasyon gibi belirli görevler için birçok kütüphane mevcuttur.

- **Paket Yöneticileri:** npm (Node Package Manager) ve Yarn gibi JavaScript paket yöneticileri, JavaScript paketlerini ve bağımlılıklarını keşfetme, kurma ve yönetme konusunda kolaylık sağlar. Bu yöneticiler, geliştiricilere projelerine üçüncü taraf kütüphane ve çerçeveleri kolayca dahil etmelerini sağlar ve kod yeniden kullanımını ve verimliliği teşvik eder.

Javascript Ekosistemi

- **Derleme Araçları ve Görev Yürütücüler:** Webpack, Babel ve Grunt gibi araçlar, geliştiricilere görevleri otomatikleştirme, kodu paketleme ve optimize etme, yeni JavaScript özelliklerini geriye dönük uyumluluk için dönüştürme ve çeşitli derleme süreçlerini yönetme imkanı sağlar. Bu araçlar, geliştirme iş akışını geliştirir ve verimliliği artırır.

- **Test ve Kalite Güvencesi:** JavaScript ekosistemi, Jest, Mocha ve Jasmine gibi geniş bir test çerçevesi ve kütüphane yelpazesi sunar. Bu test araçlarıyla JavaScript kodu için birim testleri, entegrasyon testleri ve uçtan uca testler yazılabilir. Kalite güvencesi araçları, uygulamaların kararlılığını ve güvenilirliğini sağlamak için kullanılır.

- **Topluluk ve Belgelendirme:** JavaScript ekosistemi, geniş ve aktif bir geliştirici topluluğu tarafından desteklenir. Öğrenme, bilgi paylaşma ve yardım talep etme için birçok çevrimiçi kaynak, forum ve belge mevcuttur. Açık kaynaklı projeler ve işbirlikçi geliştirme yaygındır ve sürekli yenilik ve gelişmeyi teşvik eder.

Javascript ve Ecmascript

- JavaScript, Ecmascript spesifikasyonuna dayanan bir programlama dilidir. Ecmascript, JavaScript'in temel özelliklerini ve dilin nasıl çalışması gerektiğini tanımlayan bir standarttır. JavaScript, Ecmascript spesifikasyonunun bir uygulamasıdır ve bu standartta belirtilen kurallara ve özelliklere uymaktadır.
- Ecmascript, dilin zamanla gelişmesi ve güncellenmesi için kullanılan bir standarttır. İlk olarak 1997 yılında ECMAScript 1 (ES1) olarak yayınlandı ve o zamandan beri sürümleri güncellenerek yeni özellikler ve iyileştirmeler eklenmiştir. Örneğin, ECMAScript 6 (ES6) 2015 yılında yayınlanmış olup, önemli yeni özellikler ve dilin geliştirilmesine yönelik önemli iyileştirmeler getirmiştir .Bu yeni özellikler ve iyileştirmeler, JavaScript dilinin daha güçlü, esnek ve etkili olmasını sağlar. Örneğin, ES6 ile birlikte gelen özellikler arasında okunabilirlik ve yazılabilirlik açısından iyileştirmeler, yeni veri yapıları (Set, Map), sınıflar, arrow fonksiyonları, modüller ve daha fazlası bulunur.
- JavaScript dilinde, tarayıcılar ve Node.js gibi çeşitli platformlar, Ecmascript spesifikasyonunun belirli bir sürümünü destekleyebilir veya desteklemeyebilir. Bu nedenle, JavaScript'in kullanıldığı ortamın Ecmascript sürümünü dikkate almak önemlidir. Örneğin, bazı özellikler eski tarayıcılar tarafından desteklenmeyebilir, bu yüzden bu tarayıcılarda çalıştırılacak JavaScript kodu Ecmascript uyumluluğuna göre ayarlanmalıdır.

https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

Javascript’e Giriş Scope/Closures/Hoisting

Bu bölümde Javascript dilinin temellerini ve scope, closures ve hoisting kavramlarını inceleyeceğiz.

Değişkenler

- JavaScript'te verileri saklamak için değişkenler kullanılır. Değişken tanımlamak için **var**, **let** veya **const** anahtar kelimelerini kullanabilirsiniz.

```
var name = "John";
let age = 25;
const PI = 3.14;
```

- **var:** ES5 (ECMAScript 5) standartında kullanılan bir değişken tanımlama anahtar kelimesidir. var ile tanımlanan bir değişken, işlev veya global kapsamda (scope) erişilebilir olur. Ayrıca, var ile tanımlanan bir değişken tekrar tanımlanabilir ve güncellenebilir.

```
var name = "John";
var age = 25;

name = "David"; // Değer güncellenebilir
var age = 30; // Tekrar tanımlanabilir
```

Değişkenler

- **let:** ES6 (ECMAScript 2015) standartıyla birlikte tanıtılan bir değişken tanımlama anahtar kelimesidir. let ile tanımlanan bir değişken, blok kapsamında (scope) erişilebilir olur. Yani, değişken yalnızca tanımlandığı blok içinde geçerlidir. let, var'ın global kapsama (scope) sızmasını önler ve daha sıkı bir kapsama sahiptir.

```
let name = "John";
let age = 25;
name = "David"; // Değer güncellenebilir
let age = 30; // Hata! Tekrar tanımlanamaz
```

- **const:** ES6 ile birlikte tanıtılan bir başka değişken tanımlama anahtar kelimesidir. const ile tanımlanan bir değişkenin değeri sabittir ve bir kez atandıktan sonra değiştirilemez. const ile tanımlanan değişkenler blok kapsamında (scope) erişilebilirler. Değişmez (immutable) değerler için tercih edilir ve hata yapma olasılığını azaltır.

```
const name = "John";
const age = 25;
name = "David"; // Hata! Değer değiştirilemez
const age = 30; // Hata! Tekrar tanımlanamaz
```

Veri Tipleri

- JavaScript'te farklı veri tipleri bulunur. Temel veri tipleri şunlardır: **string** (metin), **number** (sayı), **boolean** (mantıksal değer), **array** (dizi) ve **object** (nesne).

```
var name = "John"; // string
var age = 25; // number
var isStudent = true; // boolean
var numbers = [1, 2, 3, 4, 5]; // array
var person = { name: "John", age: 25 }; // object
```

- JavaScript'te veri tipleri iki ana kategoriye ayrılır: ilkel (primitive) ve ilkel olmayan (non-primitive) veri tipleri.

  İlkel Veri Tipleri: String, number, boolean, null, undefined, symbol.
  İlkel Olmayan Veri Tipleri: Object, Array, Function, Date, RegExp, ...

Veri Tipleri

- İlkel veri tipleri değerleri doğrudan bellekte saklar ve değişkenlerin değeri bir kopyası olarak atanır. Örneğin:

```
var x = 5;
var y = x;
y = 10;
console.log(x); // 5
console.log(y); // 10
```

- İlkel olmayan (non-primitive) veri tipleri ise değerleri referans olarak saklar. Yani, bir değişken aslında verinin bellekteki konumunu gösteren bir referansı tutar. Bu nedenle, bir değişkenin değeri başka bir değişkene atanırsa, referans kopyalanır, yani aynı nesneye işaret ederler. Örneğin:

```
var arr1 = [1, 2, 3];
var arr2 = arr1;
arr2.push(4);
console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

Operatörler

- JavaScript'te matematiksel, karşılaştırma ve mantıksal operatörler kullanılabilir. Örnek olarak **+, -, *, /** gibi matematiksel operatörler, **==, ===, <, >** gibi karşılaştırma operatörleri ve **&&, ll, !** gibi mantıksal operatörler kullanılabilir.

```
var x = 5;
var y = 3;
var sum = x + y; // 8
var isGreater = x > y; // true
var logicalResult = (x > 0) && (y < 10); // true
```

- JavaScript'te == ve === iki farklı karşılaştırma operatörüdür ve aralarında önemli bir fark vardır:

```
5 == 5 // true
5 == "5" // true (Tip dönüşümü yapıldı)
true == 1 // true (true, 1 olarak dönüştürüldü)
null == undefined // true
5 === "5" // false (Değerler aynı, ancak tipler farklı)
true === 1 // false (Farklı tipler)
null === undefined // false (Farklı tipler)
```

Koşullu İfadeler

- JavaScript'te **if, else if** ve **else** ifadeleriyle koşullu ifadeler oluşturabilirsiniz. Bu ifadeler, belirli bir şartın doğru olup olmadığını kontrol etmek ve farklı kod bloklarını çalıştırmak için kullanılır.

```
var hour = 10;

if (hour < 12) {
  console.log("Good morning!");
} else if (hour < 18) {
  console.log("Good afternoon!");
} else {
  console.log("Good evening!");
}

var message = hour < 12 ?
     "Good morning!" : (hour < 18 ? "Good afternoon!" : "Good evening!");
```

Döngüler

- JavaScript'te döngüler kullanarak belirli kod bloklarını tekrarlayabilirsiniz. En yaygın kullanılan döngüler **for** ve **while** döngüleridir.

Örnek for döngüsü:

```
for (var i = 0; i < 5; i++) {
  console.log(i);
}
```

Örnek while döngüsü:

```
var i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

Döngüler

- For...in Döngüsü:

```
var obj = { name: "John", age: 30, city: "New York" };
for (var key in obj) {
  console.log(key + ": " + obj[key]);
}
// Çıktı: name: John  age: 30  city: New York
```

- For...of Döngüsü:

```
var arr = [1, 2, 3, 4, 5];
for (var element of arr) {
  console.log(element);
}
// Çıktı: 1 2 3 4 5
```

Fonksiyonlar

- JavaScript'te fonksiyonlar, kodu gruplandırmak, tekrar kullanılabilirlik sağlamak ve belirli bir görevi gerçekleştirmek için kullanılan önemli yapısal unsurlardır.

```
function multiply(a, b) {
  return a * b;
}

var result = multiply(5, 3);
console.log(result); // 15

var greet = function(name) {
  console.log("Hello, " + name + "!");
};

greet("John"); // Hello, John!
```

Scope (Kapsam)

JavaScript'te her bir fonksiyonun bir kapsamı vardır. Kapsam, değişkenlerin ve fonksiyonların erişilebilir olduğu alanı belirtir. JavaScript'te iki ana kapsam tipi vardır:

- **Global Scope:** Dosya veya script düzeyinde tanımlanan değişkenlerin ve fonksiyonların global kapsamda olduğu anlamına gelir. Bu değişkenlere ve fonksiyonlara herhangi bir yerden erişilebilir.
- **Local Scope:** Bir fonksiyon içinde tanımlanan değişkenlerin ve fonksiyonların yerel kapsamda olduğu anlamına gelir. Bu değişkenler ve fonksiyonlar, sadece tanımlandıkları fonksiyon içinde erişilebilirler.

```
var globalVariable = "Global"; // Global kapsamda tanımlanan değişken
function myFunction() {
  var localVariable = "Local"; // Local kapsamda tanımlanan değişken
  console.log(localVariable); // Local
  console.log(globalVariable); // Global
}
myFunction();
console.log(localVariable); // Hata! localVariable, local kapsamın dışında
console.log(globalVariable); // Global
```

Closures (Kapanışlar)

- Closure, bir fonksiyonun dışarıdaki değişkenlere erişim yeteneğini korumasını ifade eder. Bir fonksiyon, tanımlandığı kapsamın dışında bir yerde çağrıldığında bile, kapanışlar sayesinde tanımlandığı kapsamdaki değişkenlere erişebilir.

```
function outerFunction() {
  var outerVariable = "Outer";

  function innerFunction() {
    console.log(outerVariable); // Outer
  }

  return innerFunction;
}

var closure = outerFunction();
closure(); // Inner
```

Hoisting (Yükseltme)

- Hoisting, JavaScript'in derleme aşamasında değişken ve fonksiyon tanımlamalarını, kodun en üstüne taşımasıdır. Bu, değişken ve fonksiyonları tanımlamadan önce kullanabilmenizi sağlar. Ancak, sadece tanımlamalar yukarı taşınır, atamalar taşınmaz.

```
console.log(x); // undefined
var x = 5;
myFunction(); // Hello!
function myFunction() {
  console.log("Hello!");
}
```

- Bu örnekte, var x = 5; ifadesi, hoisting sayesinde derleme aşamasında en üste taşınır. Bu nedenle, console.log(x) ifadesi çalıştırıldığında undefined çıktısı alınır. Aynı şekilde, function myFunction() ifadesi de hoisting sayesinde en üste taşınır ve myFunction() çağrıldığında "Hello!" çıktısı alınır.

JS Objects & Prototypes

Bu bölümde Javascript’te object ve prototype kavramlarını inceleyeceğiz.

Nesnelerin Tanımlanması

- JavaScript'te nesneler (objects), özellik-değer çiftlerini içeren ve karmaşık veri yapılarını temsil eden yapılardır. Nesneler, anahtarlarla indekslenmiş özelliklere erişim sağlar. JavaScript'te bir nesne tanımlamak için süslü parantezleri **({})** kullanırız. Bu süslü parantezler içinde, her özelliği bir anahtar-değer çifti olarak tanımlarız. Anahtarlar bir dize (string) olmalıdır ve değerler herhangi bir veri tipi olabilir.

```
var person = {
  name: "John",
  age: 25,
  isStudent: true
};
```

Yukarıdaki örnekte, person adında bir nesne tanımlanmıştır. Bu nesne, "name", "age" ve "isStudent" gibi üç özelliğe sahiptir.

Özelliklere Erişim

- Nesnelerdeki özelliklere nokta (.) veya köşeli parantez ([]) notasyonu kullanarak erişebiliriz. Nokta notasyonunda özelliğin adını doğrudan yazarken, köşeli parantez notasyonunda özelliğin adını bir dize (string) olarak belirtiriz.

```
console.log(person.name); // "John"
console.log(person["age"]); // 25
```

Yukarıdaki örneklerde, person nesnesindeki "name" ve "age" özelliklerine erişim sağlanmıştır.

Nesne Metodları

JavaScript'te nesnelere metod eklemek için iki farklı yol vardır: prototip tabanlı ve sınıf tabanlı yaklaşımlar. İlk olarak, prototip tabanlı yönteme bir göz atalım:

- **Prototip Tabanlı Yaklaşım:** JavaScript'te nesnelerin prototipleri vardır ve yeni metodları prototipe ekleyebiliriz. Bu, prototipte tanımlanan bir metodu tüm örneklerde kullanmamızı sağlar.

```
// Bir örnek nesne oluşturalım
var nesne = {
  isim: "Nesne",
  yazdir: function() {
    console.log(this.isim);
  }
};

// Prototipe yeni bir metod ekleyelim
nesne.yeniMetod = function() {
  console.log("Bu yeni bir metod");
};

// Metodları kullanalım
nesne.yazdir();      // Çıktı: "Nesne"
nesne.yeniMetod();   // Çıktı: "Bu yeni bir metod"
```

Nesne Prototipleri

- JavaScript'te nesne prototipleri (object prototypes), nesnelerin özellik ve davranışlarını kalıtım yoluyla paylaşmalarını sağlayan bir özelliktir. Nesne prototipleri, bir nesnenin prototipine referans veren bir özelliği kullanarak oluşturulur. Bu konu, JavaScript'teki prototip tabanlı kalıtımın temelini oluşturur.

  Prototip Kavramı:
  JavaScript'te her nesne, bir prototip ile ilişkilidir. Bir nesnenin prototipi, o nesneyi tanımlayan ve özelliklerini ve metodlarını içeren başka bir nesnedir. Nesne prototipleri, bir nesnenin özelliklerine ve metodlarına erişim sağlar ve nesne oluşturulurken kalıtım yoluyla paylaşılırlar.

  Prototip Zinciri:
  JavaScript'teki nesne prototipleri, bir prototip zinciri oluşturur. Bir nesnenin prototipi, kendisine bağlı olan prototipe (üst prototip) referans verir. Bu şekilde, bir nesne, kendisine ait olmayan özelliklere ve metodlara prototip zinciri üzerinden erişebilir.

  Prototip Tanımlama:
  Bir nesne prototipi, nesne oluşturulurken **Object.create()** veya **new** anahtar kelimeleri kullanılarak tanımlanabilir.

Nesne Prototipleri

- **prototype Özelliği:** JavaScript'te fonksiyonların bir prototype özelliği vardır. Bu özellik, o fonksiyonu kullanarak oluşturulan nesnelerin prototipi olarak kullanılabilir. Bu özellik, fonksiyon tabanlı prototip oluşturma için kullanılır.

```
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  console.log("Hello, my name is " + this.name);
};

var person = new Person("John");
person.greet(); // "Hello, my name is John"
```

Javascript DOM Yönetimi

Bu bölümde DOM üzerindeki elementlerin Javascript ile yönetilmesini işleyeceğiz.

DOM Nedir?

- DOM, bir web tarayıcısının HTML veya XML belgesini yüklediğinde, bu belgeyi bellekte temsil eden bir yapıdır.
- Örnekte, **<h1>** ve **<p>** elementleri birer DOM nesnesidir. JavaScript, DOM API'sini kullanarak bu nesnelere erişebilir ve özelliklerini değiştirebilir. Örneğin, **getElementById**() yöntemi kullanılarak "**baslik**" ID'sine sahip olan başlık elementine erişilebilir ve içeriği değiştirilebilir. DOM, web sayfalarının canlandırılması, içeriklerin dinamik olarak oluşturulması ve kullanıcı etkileşimlerinin yönetilmesi gibi birçok farklı senaryoda kullanılabilir.

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Örneği</title>
</head>
<body>
  <h1 id="baslik">Merhaba Dünya!</h1>
  <p>Bu bir örnek paragraf.</p>
</body>
</html>
```

DOM’u Manipüle Etme

- Örnekte, **getElementById**() yöntemini kullanarak bir buton elementini seçtik. Ardından, **addEventListener**() yöntemini kullanarak bu butona bir tıklama olayı dinleyicisi ekledik. Tıklama olayı tetiklendiğinde, butonun metnini değiştiriyoruz.
- Aynı şekilde, **createElement**() yöntemini kullanarak yeni bir **p** (paragraf) elementi oluşturduk ve **textContent** özelliğini ayarladık. Son olarak, bu yeni elementi **appendChild**() yöntemiyle belgenin body elementine ekledik.

```
// Örnek: Bir butona tıklandığında metni değiştirme
var button = document.getElementById("myButton");
button.addEventListener("click", function() {
  button.textContent = "Tıklandı!";
});

// Örnek: Yeni bir element oluşturma ve belgeye ekleme
var newElement = document.createElement("p");
newElement.textContent = "Bu yeni bir paragraf.";
document.body.appendChild(newElement);
```

Element Seçimi

- **getElementById**(): Bir ID'ye göre bir element seçmek için kullanılır.
  Örneğin: `var element = document.getElementById("myElement");`

- **getElementsByClassName**(): Bir sınıfa göre elementleri seçmek için kullanılır.
  Örneğin: `var elements = document.getElementsByClassName("myClass");`

- **getElementsByTagName**(): Bir etikete göre elementleri seçmek için kullanılır.
  Örneğin: `var elements = document.getElementsByTagName("div");`

- **querySelector**(): CSS seçicilerini kullanarak bir element seçmek için kullanılır.
  Örneğin: `var element = document.querySelector("#myElement .myClass");`

- **querySelectorAll**(): CSS seçicilerini kullanarak tüm uygun elementleri seçmek için kullanılır.
  Örneğin: `var elements = document.querySelectorAll("p.myClass");`

Element Özelliklerini Değiştirme

- **element.textContent:** Elementin metin içeriğini alır veya değiştirir.
```
var element = document.getElementById("myElement");
element.textContent = "Yeni metin içeriği";
```
- **element.innerHTML:** Elementin HTML içeriğini alır veya değiştirir.
```
var element = document.getElementById("myElement");
element.innerHTML = "<strong>Yeni HTML içeriği</strong>";
```
- **element.setAttribute():** Bir özniteliği ekler veya değiştirir.
```
var element = document.getElementById("myElement");
element.setAttribute("src", "yeniResim.jpg");
```
- **element.removeAttribute():** Bir özniteliği kaldırır.
```
var element = document.getElementById("myElement");
element.removeAttribute("class");
```
- **element.style:** Elementin CSS özelliklerini değiştirmek için kullanılır.
```
var element = document.getElementById("myElement");
element.style.color = "red";
element.style.fontSize = "20px";
```

Yeni Element Oluşturmak ve Eklemek

- örnekte, **createElement**() yöntemi kullanılarak yeni bir ** ** elementi oluşturulur. Ardından **textContent** özelliği kullanılarak oluşturulan div elementine bir metin içeriği atanır. Son olarak, **appendChild**() yöntemi kullanılarak div elementi belgeye (body elementine) eklenir.

```
// Yeni bir div elementi oluşturma
var yeniDiv = document.createElement("div");

// Oluşturulan div elementine bir metin içeriği atama
yeniDiv.textContent = "Bu yeni bir div elementidir.";

// Div elementini belgeye ekleyelim
document.body.appendChild(yeniDiv);
```

Element Silmek

- Örnekte, **getElementById**() yöntemi kullanarak bir elementi seçtik.
- Ardından, **remove**() yöntemini kullanarak seçilen elementi belgeden kaldırdık.
- **remove**() yöntemi, seçilen elementi doğrudan kaldırır.

Not: Elementin ebeveynini bildiğiniz durumlarda, alternatif olarak **parentNode.removeChild(element)** yöntemini de kullanabilirsiniz. Bu yöntemde **parentNode** ile elementin ebeveynini seçip, **removeChild**() yöntemini kullanarak elementi ebeveyninden kaldırırsınız.

```
// Bir elementi belgeden kaldırma
var element = document.getElementById("myElement");
element.remove();
```

```
// Bir elementi belgeden kaldırma (parentNode.removeChild)
var parentElement = document.getElementById("parentElement");
var childElement = document.getElementById("childElement");

parentElement.removeChild(childElement);
```

Typescript

Bu bölümde Typescript genel konularını inceleyeceğiz.

Typescript Nedir ?

- TypeScript, Microsoft tarafından geliştirilen bir programlama dili ve açık kaynaklı bir projedir.
- JavaScript tabanlı bir dil olup, JavaScript'in üzerine ek özellikler ve tip sistemi getirir.
- TypeScript, statik tipleme özelliği ile JavaScript kodunun daha güvenli ve ölçeklenebilir olmasını sağlar.
- TypeScript, JavaScript kodunun derlenip JavaScript'e dönüştürülerek çalışır.
- TypeScript'in derleme süreci, kodu daha anlaşılır hale getirmek ve hataları derleme aşamasında yakalamak için kullanılır.
- JavaScript kütüphanelerini ve çerçevelerini kullanmaya olanak sağlar ve JavaScript ekosistemiyle uyumludur.
- TypeScript, büyük ölçekli projelerde daha kolay bir şekilde çalışmayı ve daha iyi bir kod yönetimi sağlamayı hedefler.
- Geliştiricilere kod tamamlama, hata tespiti, refactoring desteği ve daha fazlasını sunan gelişmiş bir entegre geliştirme ortamı (IDE) deneyimi sağlar.
- TypeScript, JavaScript kodunu daha yapılandırılmış hale getirmek ve geliştirici produktivitesini artırmak için tasarlanmıştır.
- TypeScript, ECMAScript standartlarına uyumlu olup, sürekli olarak geliştirilmekte ve yaygın olarak kullanılmaktadır.

JS ile TS Arasındaki Farklar ve Avantajlar

- **Tip Sistemleri:** TypeScript, statik tipli bir dildir. Değişkenlerin, fonksiyonların ve nesnelerin türlerini belirleyebilirsiniz. Bu, hataların daha erken yakalanmasını ve daha güvenli kod yazmanızı sağlar.
- **Sınıflar ve Nesne Tabanlı Programlama:** TypeScript, sınıfları ve nesne tabanlı programlamayı destekler. JavaScript'te bu özellikler daha sınırlıdır.
- **Yeni ECMAScript Özellikleri:** TypeScript, ECMAScript standartlarının yeni özelliklerini önceden destekler. Böylece, daha yeni JavaScript dil özelliklerini kullanabilirsiniz.
- **Geliştirme Aracı Desteği:** TypeScript, geliştirme sürecini kolaylaştıran bir dizi araç ve entegrasyon sunar. Örneğin, kod tamamlama, hata kontrolü ve refactoring için IDE'lerde destek sağlar.

TypeScript'in Avantajları:

- **Daha Güvenli Kod:** Statik tiplendirme, hataların daha erken yakalanmasını ve daha güvenli bir kod tabanı oluşturmanızı sağlar.
- **Geliştirme Sürekliliği:** TypeScript, büyük ve karmaşık projelerde daha sürdürülebilir bir geliştirme süreci sunar.
- **Geliştirici Üretkenliği:** Türlendirmenin sunduğu kod tamamlama ve hata kontrolü, geliştirici üretkenliğini artırır.
- **Uyumluluk**: TypeScript, JavaScript ile uyumlu olduğu için mevcut JavaScript kodunu kullanabilir ve mevcut kütüphaneleri entegre edebilirsiniz.
- **Güçlü Ekosistem:** TypeScript, geniş bir topluluk tarafından desteklenir ve geliştirme araçlarına sahiptir.

JS ile TS Arasındaki Farklar ve Avantajlar

- **Tip Sistemleri:** TypeScript, statik tipli bir dildir. Değişkenlerin, fonksiyonların ve nesnelerin türlerini belirleyebilirsiniz. Bu, hataların daha erken yakalanmasını ve daha güvenli kod yazmanızı sağlar.
- **Sınıflar ve Nesne Tabanlı Programlama:** TypeScript, sınıfları ve nesne tabanlı programlamayı destekler. JavaScript'te bu özellikler daha sınırlıdır.
- **Yeni ECMAScript Özellikleri:** TypeScript, ECMAScript standartlarının yeni özelliklerini önceden destekler. Böylece, daha yeni JavaScript dil özelliklerini kullanabilirsiniz.
- **Geliştirme Aracı Desteği:** TypeScript, geliştirme sürecini kolaylaştıran bir dizi araç ve entegrasyon sunar. Örneğin, kod tamamlama, hata kontrolü ve refactoring için IDE'lerde destek sağlar.

TypeScript'in Avantajları:

- **Daha Güvenli Kod:** Statik tiplendirme, hataların daha erken yakalanmasını ve daha güvenli bir kod tabanı oluşturmanızı sağlar.
- **Geliştirme Sürekliliği:** TypeScript, büyük ve karmaşık projelerde daha sürdürülebilir bir geliştirme süreci sunar.
- **Geliştirici Üretkenliği:** Türlendirmenin sunduğu kod tamamlama ve hata kontrolü, geliştirici üretkenliğini artırır.
- **Uyumluluk**: TypeScript, JavaScript ile uyumlu olduğu için mevcut JavaScript kodunu kullanabilir ve mevcut kütüphaneleri entegre edebilirsiniz.
- **Güçlü Ekosistem:** TypeScript, geniş bir topluluk tarafından desteklenir ve geliştirme araçlarına sahiptir.

Veri Tipleri

- **Int:** Tam sayı değerlerini temsil eder.
  Örnek: `let sayi: number = 10;`
- **String:** Metin verilerini temsil eder.
  Örnek: `let isim: string = "John";`
- **Boolean**: İki değeri temsil eder: true veya false.
  Örnek: `let aktif: boolean = true;`
- **Diziler:** Aynı türden birden çok değeri içeren koleksiyonlardır.
  Örnek: `let sayilar: number[] = [1, 2, 3, 4, 5];`
- **Obje:** Karmaşık veri yapılarını temsil eder.
  Örnek:
```
let kullanici: { ad: string, yas: number } = {
  ad: "John",
  yas: 30
};
```
- **Diğer Tipler:** **Enum:** Belirli değerlere sahip bir tür oluşturur.
  Any: Herhangi bir türü temsil eder, tür denetimini devre dışı bırakır.
  Void: Herhangi bir değeri temsil etmez.
  Null ve Undefined: Belirli olmayan veya tanımsız değerleri temsil eder.

Fonksiyonlar

1. Parametre ve Dönüş Tipi Belirtimi

- TypeScript, fonksiyonların parametre ve dönüş tiplerini belirtebilmemizi sağlar.
- Parametre tipleri, gelen argümanların uygunluğunu kontrol etmeye yardımcı olur.
- Dönüş tipi belirtimi ise fonksiyonun geri döndüreceği değerin tipini belirtir.

```
function toplama(a: number, b: number): number {
  return a + b;
}
```

Yukarıdaki örnekte, toplama adında bir fonksiyon tanımladık. Parametre tipleri number olarak belirtilmiştir ve fonksiyonun dönüş tipi de number olarak belirtilmiştir. Bu sayede fonksiyonun doğru argümanlarla çağrıldığından emin olabiliriz.

Fonksiyonlar

2. İsimsiz Fonksiyonlar ve Arrow Fonksiyonları

- TypeScript, isimsiz fonksiyonları ve arrow fonksiyonlarını destekler.
- İsimsiz fonksiyonlar, fonksiyonları bir değişkene veya bir nesne özelliğine atayarak kullanmamızı sağlar.
- Arrow fonksiyonları ise daha kısa bir sözdizimine sahiptir ve this bağlamını otomatik olarak korur.

```
let toplama = function(a: number, b: number): number {
  return a + b;
};
```

veya

```
let toplama = (a: number, b: number): number => a + b;
```

Sınıflar

- Sınıflar, nesne tabanlı programlamada temel yapı taşlarıdır.
- İlgili verileri ve işlevleri gruplandırır ve birlikte çalışmasını sağlar.
- Sınıf tanımları, nesnelerin özelliklerini ve davranışlarını belirler.

```
class Araba {
  marka: string;
  model: string;
  yil: number;

  constructor(marka: string, model: string, yil: number) {
    this.marka = marka;
    this.model = model;
    this.yil = yil;
  }

  bilgileriYazdir() {
    console.log(`Marka: ${this.marka}, Model: ${this.model}, Yıl: ${this.yil}`);
  }
}

// Araba sınıfından örnek oluşturma
let araba1 = new Araba("Toyota", "Corolla", 2021);
araba1.bilgileriYazdir(); // Çıktı: Marka: Toyota, Model: Corolla, Yıl: 2021
```

Sınıflar

Sınıfın Kullanımı:

- Sınıfı kullanarak bir veya daha fazla nesne örneği oluşturabiliriz.
- Nesneler, sınıfın özelliklerine ve metodlarına erişebilir ve bunları kullanabilir.
- Örnekte, Araba sınıfından bir örnek oluşturarak araba bilgilerini yazdırdık.

```
// Araba sınıfından örnek oluşturma
let araba1 = new Araba("Toyota", "Corolla", 2021);
araba1.bilgileriYazdir(); // Çıktı: Marka: Toyota, Model: Corolla, Yıl: 2021
```

Sınıfların Özellikleri

- Sınıfın Durumunu Tanımlayan Değişkenler:
  Örnek: Bir Araba Sınıfı
  Araba'nın Durumunu Tanımlayan Değişkenler
  Özelliklerin İsimlendirme Standartları
- Getter ve Setter Metodları:
  Örnek: Araba'nın Hızı
  Hız Değişkeninin Okunması ve Değiştirilmesi
  Getter ve Setter Metodlarının Tanımlanması
- Erişim Belirteçleri, Erişim Belirteçleri Nedir?
  public Erişim Belirteci
  private Erişim Belirteci
  protected Erişim Belirteci

```
class Araba {
  private _hiz: number;

  get hiz(): number {
    return this._hiz;
  }

  set hiz(yeniHiz: number) {
    if (yeniHiz >= 0) {
      this._hiz = yeniHiz;
    }
  }
}

let araba = new Araba();
araba.hiz = 100; // Setter metodunu kullanarak hızı ayarla
console.log(araba.hiz); // Getter metodunu kullanarak hızı oku
```

Sınıflarda Kalıtım

Sınıflar Arasında İlişkilerin Kurulması:

- Kalıtım ile Nesneler Arasında Bir İlişki Kurma
- Ana Sınıf ve Alt Sınıflar Arasındaki İlişki

Ana Sınıf ve Alt Sınıfların Oluşturulması:

- Ana Sınıfın Tanımlanması ve Özellikleri
- Alt Sınıfların Ana Sınıftan Miras Alma

extend Anahtar Kelimesi ve Kalıtım Yapısı:

- extend Anahtar Kelimesinin Kullanımı
- Alt Sınıfın Ana Sınıfı Genişletmesi

```
class Hayvan {
  constructor(ad) {
    this.ad = ad;
  }

  hareketEt() {
    console.log(this.ad + " hareket ediyor.");
  }
}

class Kedi extends Hayvan {
  miyavla() {
    console.log(this.ad + " miyavlıyor.");
  }
}

// Kedi sınıfının örneğini oluşturalım
let minnoş = new Kedi("Minnoş");

// Kalıtım sayesinde hem Hayvan sınıfının hem de Kedi sınıfının metotlarını
minnoş.hareketEt();   // Çıktı: "Minnoş hareket ediyor."
minnoş.miyavla();     // Çıktı: "Minnoş miyavlıyor."
```

Sınıflarda Arayüzler (Interfaces)

Bu örnek kodda, **Shape** ve **Color** adlı iki arayüz tanımlanmıştır. Ardından **Rectangle** adlı bir sınıf, **Shape** ve **Color** arayüzlerini uygulamaktadır. Sınıfın içinde **calculateArea**() ve **display**() metodları bulunurken, **color** özelliği de **Color** arayüzünden gelmektedir.

Slayt içeriği, arayüzlerin nasıl tanımlanacağını, nasıl uygulanacağını ve birden fazla arayüzün nasıl kullanılacağını açıklar. Örnek kod, bir dikdörtgenin alanını hesaplayan ve özellikleriyle birlikte ekrana yazdıran basit bir senaryoyu göstermektedir.

```
interface Shape {
  calculateArea(): number;
  display(): void;
}

interface Color {
  color: string;
}

class Rectangle implements Shape, Color {
  width: number;
  height: number;
  color: string;

  constructor(width: number, height: number, color: string) {
    this.width = width;
    this.height = height;
    this.color = color;
  }

  calculateArea(): number {
    return this.width * this.height;
  }

  display(): void {
    console.log(`This is a ${this.color} rectangle with an area of ${this.calculateArea()} square units.`);
  }
}

let rectangle = new Rectangle(5, 3, "blue");
rectangle.display();
```

Modüller (Import / Export)

Bu örnek, bir modülün nasıl dışa aktarıldığını ve başka bir modülde nasıl içe aktarıldığını göstermektedir. İlk olarak, **ad** ve **selamla değişkeni** ve **fonksiyonu** export anahtar kelimesiyle dışa aktarılır.

Ardından, başka bir modülde bu değişkeni ve fonksiyonu import anahtar kelimesiyle içe aktarırız. İçe aktarılan değişkeni ve fonksiyonu kullanarak çıktılar alabiliriz.

```
// Bir modülün dışa aktarılması
export const ad = "John Doe";
export function selamla() {
  console.log("Merhaba!");
}

// Başka bir modülde bu modülü içe aktarma
import { ad, selamla } from "./modul";

console.log(ad);        // Çıktı: "John Doe"
selamla();              // Çıktı: "Merhaba!"
```

Hata Yakalama

- try bloğunda hata olabilecek kod parçacığı yer alır.
- Eğer hata oluşursa, catch bloğu çalışır.
- catch bloğunda hata nesnesi yakalanır ve işlemler yapılır.

```
try {
  // Hata olabilecek kod bloğu
} catch (hata) {
  // Hata durumunda yapılacak işlemler
}
```

Özel Hata Sınıfları

- Özel bir istisna sınıfı tanımlanabilir.
- Error sınıfından türetilir ve istisna davranışını özelleştirebilir.
- Özel hata sınıfının adı ve ek özellikleri belirlenebilir.

```
class OzelHata extends Error {
  constructor(mesaj) {
    super(mesaj);
    this.name = "OzelHata";
  }
}
```

Hata Yakalama Örneği

- Kod örneğinde, bol fonksiyonunda bölme işlemi yapılırken b değeri sıfır ise bir hata fırlatılır.
- try bloğu içinde bol fonksiyonu çağrılır ve bir hata oluşması durumunda catch bloğu çalışır.
- Hata yakalandığında, hata nesnesi kullanılarak hata mesajı yazdırılır.

```
function bol(a, b) {
  if (b === 0) {
    throw new Error("Bölme işlemi sıfıra bölme hatası!");
  }
  return a / b;
}

try {
  let sonuc = bol(10, 0);
  console.log("Sonuç:", sonuc);
} catch (hata) {
  console.log("Hata Yakalandı:", hata.message);
}
```

Angular’a Giriş

Bu bölümde Angular ve temel özelliklerini inceleyeceğiz.

Angular Nedir ?

- Angular, Google tarafından geliştirilen bir TypeScript tabanlı bir uygulama geliştirme çerçevesidir.
- Web uygulamalarını geliştirmek için kullanılır.
- MVC (Model-View-Controller) tasarım desenine dayalı olarak çalışır.

Angular'ın Avantajları

- Modüler bir yapıya sahiptir, yani kodunuzu parçalara ayırmanızı sağlar.
- Büyük ölçekli uygulamaları yönetmek için idealdir.
- Yeniden kullanılabilir bileşenler (component) kullanarak verimli kod yazmayı sağlar.
- Hızlı ve zengin bir kullanıcı deneyimi sunar.
- TypeScript tabanlıdır, bu da daha güvenli ve daha anlaşılır bir kod yazmanızı sağlar.

Angular Temel Özellikleri

- Angular, bileşen tabanlı bir yapı sunar. Uygulama, birbiriyle etkileşimde bulunan ve yeniden kullanılabilir bileşenlerden oluşur.
- Her bileşen, kendi veri modelini, görüntüsünü ve iş mantığını içerir. Bileşenler, birleşerek zengin ve karmaşık kullanıcı arayüzleri oluşturur.
- Angular, "Data Binding" olarak adlandırılan bir özelliği destekler. Bu özellik sayesinde bileşenlerin görüntüsü, veri modeliyle senkronize bir şekilde güncellenir.
- Veri modelinde yapılan değişiklikler, otomatik olarak görüntüde yansıtılır ve kullanıcı etkileşimi sonucunda görüntüdeki değişiklikler de veri modeline yansır.
- Angular, güçlü bir routing mekanizması sunar. Bu sayede farklı sayfalar arasında gezinme, URL parametreleri ile çalışma ve derin bağlantıları destekleme gibi özellikleri kullanabilirsiniz.
- Routing, kullanıcı deneyimini geliştirmek ve uygulamanın farklı bölümlerine erişimi kolaylaştırmak için önemli bir modüldür.
- Angular, zengin bir ekosisteme sahiptir. Birçok hazır bileşen, kütüphane ve eklenti mevcuttur. Bunlar, geliştirme sürecini hızlandırır ve daha iyi bir kullanıcı deneyimi sağlar.

Single Page Application (SPA)

- Bir "Single Page Application" (SPA), web uygulamalarının geleneksel çok sayfalı yapı yerine tek bir sayfa üzerinden çalıştığı bir tür web uygulamasıdır. SPA'lar, web tarayıcısının sayfayı yeniden yükleme ihtiyacını ortadan kaldırarak, kullanıcı deneyimini geliştirmeyi amaçlar.

- Bir SPA, başlangıçta tüm gerekli kaynakları (HTML, CSS, JavaScript, vb.) indirir ve sayfa yüklendiğinde bu kaynakları kullanarak kullanıcıyla etkileşime geçer. SPA'lar genellikle JavaScript çerçeveleri veya kütüphaneleri (örneğin Angular, React veya Vue.js gibi) kullanılarak geliştirilir.

- SPA'lar, kullanıcı etkileşimleri sonucunda veri taleplerini sunucuya Ajax veya Fetch API gibi teknolojiler aracılığıyla gönderir ve sunucu, sadece gerekli veriyi geri döndürür. Bu, sayfa yeniden yüklemeleri olmadan, hızlı ve akıcı bir kullanıcı deneyimi sağlar. SPA'lar genellikle tek bir ana sayfada çalışır ve kullanıcı etkileşimleri sonucunda içerik dinamik olarak değiştirilir.

Angular CLI

- Angular CLI (Command Line Interface), Angular uygulamalarının hızlı bir şekilde oluşturulmasına ve geliştirilmesine yardımcı olan bir komut satırı aracıdır. Angular CLI, bir dizi kullanışlı komut ve araçlar sağlar ve geliştirme sürecini kolaylaştırır.

- Angular CLI, Angular projelerini oluşturmak, bileşenler, hizmetler, yönlendiriciler ve diğer yapı taşları oluşturmak gibi yaygın görevleri hızlı bir şekilde gerçekleştirmek için kullanılır. Ayrıca, derleme, hata ayıklama, test çalıştırma ve uygulamanın geliştirme sunucusunu başlatma gibi işlemleri de yönetir.

Angular CLI

Angular CLI kullanmak için aşağıdaki adımları izleyebilirsiniz:

- Angular CLI'yı yükleyin: Angular CLI'nın yüklü olması gereklidir. CLI'yi yüklemek için Node.js'in yüklü olduğu bir ortama sahip olmanız ve bir komut satırı aracı kullanarak npm install -g @angular/cli komutunu çalıştırmanız gerekir.

- Yeni bir Angular projesi oluşturun: Bir komut satırı aracı kullanarak ng new proje-adi komutunu çalıştırın. Bu, yeni bir Angular projesi oluşturacak ve gerekli dosyaları ve klasörleri oluşturacaktır.

- Proje dizinine gidin: Oluşturulan proje klasörüne gidin (cd proje-adi) ve projeyi düzenlemeye başlayın.

- Yapı taşları oluşturun: CLI, bileşenler, hizmetler, yönlendiriciler ve diğer yapı taşlarını hızlıca oluşturmanızı sağlar. Örneğin, yeni bir bileşen oluşturmak için ng generate component bileşen-adi komutunu kullanabilirsiniz.

- Proje geliştirme sunucusunu başlatın: Geliştirme sürecinde uygulamanızı test etmek için yerel bir sunucu başlatmanız gerekebilir. Bunun için ng serve komutunu kullanabilirsiniz. Bu komut, uygulamanızı derler ve yerel bir sunucuda çalıştırır. Tarayıcınızdan http://localhost:4200 adresine giderek uygulamayı görüntüleyebilirsiniz.

Angular Proje ve Klasör Yapısı

Angular projeleri genellikle aşağıdaki gibi bir dosya ve klasör yapısına sahiptir:

- **src** klasörü: Bu klasör, projenin kaynak dosyalarını içerir. Çoğu geliştirme faaliyeti bu klasör altında gerçekleştirilir. Aşağıdaki dosya ve klasörleri içerebilir:
    - **app** klasörü: Uygulamanın bileşenlerini, hizmetlerini, yönlendiricilerini ve diğer yapı taşlarını içeren klasördür. Bu klasörde çalışma yapısını düzenlemek için alt klasörler oluşturabilirsiniz.
    - **assets** klasörü: Uygulamanın kullanacağı statik dosyaları (görüntüler, fontlar, JSON dosyaları vb.) içerir.
    - **environments** klasörü: Farklı ortamlar için yapılandırma dosyalarını içeren klasördür. Örneğin, geliştirme ortamı ve üretim ortamı için ayrı yapılandırma dosyaları olabilir.
    - **index.html** dosyası: Uygulamanın temel HTML dosyasıdır. Angular bileşenleri bu dosya üzerinde çalışır.
    - **styles.scss** veya **styles.css** dosyası: Uygulamanın genel stil kurallarını içeren dosyadır. Proje yapılandırmasına bağlı olarak Sass veya CSS kullanılabilir.
    - **main.ts** dosyası: Uygulamanın başlangıç noktasıdır. Angular uygulamasının başlatılmasını sağlar.

Angular Proje ve Klasör Yapısı

Angular projeleri genellikle aşağıdaki gibi bir dosya ve klasör yapısına sahiptir:

- **angular.json** dosyası: Projenin derleme, yapılandırma ve paketleme ayarlarını içeren JSON formatında bir yapılandırma dosyasıdır. Bu dosya, projenin derlenmiş çıktısı, statik dosyaların nasıl yönetileceği, yapı taşlarının nasıl oluşturulacağı vb. gibi birçok ayarı içerir.
- **package.json** dosyası: Projenin bağımlılıklarını ve projenin komutlarını içeren dosyadır. Bu dosyada kullanılan paketlerin sürümleri, proje komutları ve diğer proje ayarları yer alır.
- **node_modules** klasörü: Proje için kullanılan tüm bağımlılıkların ve paketlerin bulunduğu klasördür. Bu klasör, npm install veya yarn install gibi komutlarla oluşturulur.
- **tsconfig.json** dosyası: TypeScript derleme ayarlarını içeren dosyadır. TypeScript dosyalarının nasıl derleneceği, dışa aktarımların nasıl yapılacağı ve diğer derleme ayarları bu dosyada tanımlanır.
- **karma.conf.js** veya **jest.config.js** dosyası: Unit testlerin yapılandırma ayarlarını içeren dosyadır. Angular projeleri genellikle Karma veya Jest kullanarak testlerini yönetir.

Angular Components

Bu bölümde Angular component (bileşen) yapısını inceleyeceğiz.

Component Oluşturulması

Angular CLI kullanarak component oluşturabilir ve kullanıma hazır hale getirebiliriz.

- Terminali açın ve Angular projesinin bulunduğu dizine gidin.
- Bileşenin oluşturulacağı komutu çalıştırın:
  `ng generate component header`
- Angular CLI, bileşenin dosyalarını ve klasörünü otomatik olarak oluşturur.
- Bileşeni kullanmak için, oluşturulan bileşenin adını başka bir bileşenin şablonunda veya TypeScript kodunda kullanabilirsiniz.

Böylece bileşen oluşturma adımlarını takip ederek Angular'da yeni bileşenler oluşturabilirsiniz. Bileşenler, tekrar kullanılabilir ve modüler bir yapıya sahip olup Angular'da kullanıcı arayüzünü oluşturmanın temel birimidir.

Component Kullanımı

Bir bileşenin kullanımı ve bileşen örneklerinin kullanımı, Angular'da kullanıcı arayüzünü oluşturmak için önemlidir.

- Oluşturulan bileşen, kullanılacağı modüle eklenmelidir. Bunun için, bileşenin kullanılacağı modül dosyasını açın ve bileşeni ekleyin. Bunu declarations dizisine veya gerekli olduğunda exports dizisine ekleyebilirsiniz.
- Bileşenin kullanıcı arayüzü, bileşenin şablonunda tanımlanır. Şablon, bileşenin HTML kodunu içerir. Bileşenin kullanıldığı yerde, bileşenin şablonunu kullanmak için bileşen etiketini kullanmanız gerekmektedir. Örneğin: **`<app-component-name></app-component-name>`**

Bu şekilde, oluşturduğunuz bileşenleri kullanabilir ve bileşenler arasında veri alışverişi yapabilirsiniz. Bileşenler, Angular'da yeniden kullanılabilirlik ve modülerlik sağlamak için önemli bir yapı taşıdır.

Stillendirme

Angular'da stil direktifleri, bileşenlerin görünümünü ve stili kontrol etmek için kullanılır.

- **ngClass**:
  ngClass direktifi, bileşene dinamik olarak CSS sınıfları eklemek veya kaldırmak için kullanılır.
  Bileşenin şablonunda kullanılacak olan ngClass direktifi, bir obje veya dizi alabilir.
  Örneğin, aşağıdaki kodda, bir div elementi isActive özelliği true olduğunda "active" sınıfını alır:
  ```
  <div [ngClass]="{ 'active': isActive }">Some Content</div>
  ```
- **ngStyle**:
  ngStyle direktifi, bileşenin stilini dinamik olarak değiştirmek için kullanılır.
  Bileşenin şablonunda kullanılacak olan ngStyle direktifi, bir obje olarak tanımlanır ve stil özelliklerini içerir.
  Örneğin, aşağıdaki kodda, bir div elementinin arka plan rengi ve yazı rengi isActive özelliğine bağlı olarak değişir:
  ```
  <div [ngStyle]="{ 'background-color': isActive ? 'red' : 'blue', 'color': isActive ? 'white' : 'black' }">Some Content</div>
  ```

Component Sınıfı

- Bileşen sınıfına bileşen özelliklerini (properties) ekleyin. Bunlar, bileşenin içinde kullanacağınız değişkenlerdir ve bileşenin durumunu temsil ederler. Örneğin:

```
export class MyComponent {
  title: string = 'Merhaba Dünya';
  count: number = 0;
  isEnabled: boolean = true;
}
```

- Bileşen özelliklerini bileşen şablonunda kullanmak için bileşen sınıfınızın içindeki özellikleri şablon dosyasına aktarmanız gerekir. Bunun için Angular'ın veri bağlama mekanizmasını kullanabilirsiniz.
  Örneğin, title özelliğini bileşenin şablonunda kullanmak için aşağıdaki şekilde yapabilirsiniz:
  `<h1>{{ title }}</h1>`

Component Sınıfı

- Bileşen sınıfına bileşen özelliklerini (properties) ekleyin. Bunlar, bileşenin içinde kullanacağınız değişkenlerdir ve bileşenin durumunu temsil ederler. Örneğin:

```
export class MyComponent {
  title: string = 'Merhaba Dünya';
  count: number = 0;
  isEnabled: boolean = true;
}
```

- Bileşen özelliklerini bileşen şablonunda kullanmak için bileşen sınıfınızın içindeki özellikleri şablon dosyasına aktarmanız gerekir. Bunun için Angular'ın veri bağlama mekanizmasını kullanabilirsiniz.
  Örneğin, title özelliğini bileşenin şablonunda kullanmak için aşağıdaki şekilde yapabilirsiniz:
  `<h1>{{ title }}</h1>`

Input() ve Output()

Angular bileşen sınıfında @Input() ve @Output() dekoratörlerini kullanarak giriş (input) ve çıkış (output) özelliklerini tanımlayabilirsiniz.

- **@Input()** dekoratörü, bileşene dışarıdan veri aktarılmasını sağlar. Bu özellikle, ebeveyn bileşenden gelen verileri almak için kullanılır. İşte bir @Input() örneği:

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: '<p>{{ inputData }}</p>'
})
export class ChildComponent {
  @Input() inputData: string;
}
```

Input() ve Output()

- **@Output()** dekoratörü, bileşenden dışarıya veri aktarılmasını sağlar. Bu özellikle, bileşenin ebeveyn bileşeniyle iletişim kurmasını sağlar. İşte bir @Output() örneği:

```
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `
    <button (click)="handleClick()">Click me</button>
  `
})
export class ChildComponent {
  @Output() outputData: EventEmitter<string> = new EventEmitter<string>();

  handleClick() {
    this.outputData.emit('Data to be sent');
  }
}
```

Veri Bağlama

- **Tek yönlü veri bağlama (interpolation) kullanımı**
  Tek yönlü veri bağlama (interpolation), Angular bileşenleri içinde verilerin bileşen sınıfından şablon dosyasına aktarılması için kullanılan bir yöntemdir. Şablon dosyasında çift süslü parantez ({{}}) kullanılarak gerçekleştirilir.

- **Özellik bağlama (property binding) ve olay bağlama (event binding)**
  - Özellik bağlama, bileşen sınıfındaki bir özelliğin, şablon dosyasındaki bir HTML öğesine bağlanmasıdır.
  - Bir özelliğin değeri, bileşen sınıfından şablondaki bir HTML öğesine aktarılır ve şablonun görünümünü günceller.
  - Özellik bağlamak için köşeli parantez [ ] kullanılır.
  - Özelliğin değeri genellikle bileşen sınıfında bir değişken veya bir metodun sonucu olabilir.
  - Özellik bağlama, bileşen sınıfında yapılan değişikliklerin şablonda anında yansımasını sağlar.

- **Çift yönlü veri bağlama (two-way binding) ve [(ngModel)] kullanımı**
  Çift yönlü veri bağlama (two-way binding), Angular'da bir bileşen özelliğini hem göstermek hem de kullanıcı tarafından yapılan değişiklikleri bileşene yansıtmak için kullanılan bir tekniktir. [(ngModel)] ise Angular'ın bu çift yönlü veri bağlamayı kolaylaştırmak için sağladığı bir yapıdır.

Veri Bağlama

- **Tek yönlü veri bağlama (interpolation) kullanımı**
  Tek yönlü veri bağlama (interpolation), Angular bileşenleri içinde verilerin bileşen sınıfından şablon dosyasına aktarılması için kullanılan bir yöntemdir. Şablon dosyasında çift süslü parantez ({{}}) kullanılarak gerçekleştirilir.

- **Özellik bağlama (property binding) ve olay bağlama (event binding)**
  - Özellik bağlama, bileşen sınıfındaki bir özelliğin, şablon dosyasındaki bir HTML öğesine bağlanmasıdır.
  - Bir özelliğin değeri, bileşen sınıfından şablondaki bir HTML öğesine aktarılır ve şablonun görünümünü günceller.
  - Özellik bağlamak için köşeli parantez [ ] kullanılır.
  - Özelliğin değeri genellikle bileşen sınıfında bir değişken veya bir metodun sonucu olabilir.
  - Özellik bağlama, bileşen sınıfında yapılan değişikliklerin şablonda anında yansımasını sağlar.

- **Çift yönlü veri bağlama (two-way binding) ve [(ngModel)] kullanımı**
  Çift yönlü veri bağlama (two-way binding), Angular'da bir bileşen özelliğini hem göstermek hem de kullanıcı tarafından yapılan değişiklikleri bileşene yansıtmak için kullanılan bir tekniktir. [(ngModel)] ise Angular'ın bu çift yönlü veri bağlamayı kolaylaştırmak için sağladığı bir yapıdır.

Lifecycle Hooks

Bu bölümde Angular component yapısındaki yaşam döngüsü metodlarını inceleyeceğiz.

Component Yaşam Döngüsü

Angular bileşenlerinin yaşam döngüsü olayları, bileşenin oluşturulması, güncellenmesi ve yok edilmesi gibi belirli anlarda tetiklenen olaylardır. Bu olayları tanımlayabilir ve bileşeninizin bu olaylara tepki vermesini sağlayabilirsiniz.

HOOK ORDER	LOG MESSAGE
1	`OnChanges`
2	`OnInit`
3	`DoCheck`
4	`AfterContentInit`
5	`AfterContentChecked`
6	`AfterViewInit`
7	`AfterViewChecked`
8	`DoCheck`
9	`AfterContentChecked`
10	`AfterViewChecked`
11	`OnDestroy`

ngOnChanges

Bileşenin giriş özelliklerinde (input properties) bir değişiklik olduğunda tetiklenir. Bu olay, bileşene gelen verinin değiştiği durumlarda kullanılır. ngOnChanges yöntemi, SimpleChanges nesnesini parametre olarak alır ve değişen özellikleri izlemenize ve işlemenize olanak tanır.

```
ngOnChanges(changes: SimpleChanges) {
  // Değişen özellikleri işle
  // changes.propName.currentValue ile yeni değere erişebilirsiniz
}
```

ngOnInit

Bileşen oluşturulduğunda, diğer yaşam döngüsü olaylarından önce tetiklenir. Bu olay, bileşenin başlangıç durumunu ayarlamak veya verileri yüklemek gibi işlemler için kullanılır.

```
ngOnInit() {
  // Bileşenin başlangıç işlemlerini yap
}
```

ngAfterViewInit

Bileşenin görünüm ağacı oluşturulduktan ve görünüm öğeleri oluşturulduktan sonra tetiklenir. Bu olay, bileşenin görünümle ilgili işlemler yapması gereken durumlar için kullanılır.

```
ngAfterViewInit() {
  // Görünümle ilgili işlemleri yap
}
```

ngOnDestroy

Bileşen yok edildiğinde tetiklenir. Bu olay, bileşenin kaynakları serbest bırakması veya temizlik işlemleri yapması gereken durumlar için kullanılır.

```
ngOnDestroy() {
  // Kaynakları serbest bırak ve temizlik işlemlerini yap
}
```

Angular Template

Bu bölümde Angular component içindeki şablon yapısını inceleyeceğiz.

Teşekkürler! :)