JavaScript Nedir?

Genellikle web tarayıcılarında kullanılan yüksek düzeyli, yorumlanabilir bir programlama dilidir. JavaScript, dinamik içerik oluşturmak ve web sayfalarını etkileşimli hale getirmek için kullanılır. JavaScript, HTML ve CSS ile birlikte, modern web geliştirmenin üç temel teknolojisinden biridir.

Yüksek düzeyli programlama dili, insanların anlaması ve kullanması daha kolay olan, genellikle belirli bir programlama modeline veya uygulama alanına yönelik olan bir programlama dilidir. Bu diller, düşük düzeyli dillerin aksine, donanım detaylarından soyutlanmıştır ve genellikle daha okunabilir ve anlaşılır bir sözdizimine sahiptir.

Örnek olarak, JS gibi yorumlanabilir bir dildir ve kodunuzu yürütmek için bir yorumlayıcı kullanır. Bu, JS kodunun yazılım geliştiricileri tarafından yazıldığında kolayca anlaşılabilir olmasını sağlar. Diğer yandan, derlenmiş dillerde (örneğin, C++), kod önce bir derleyici tarafından makine diline çevrilir ve ardından çalıştırılır. Bu tür dillerin kodu daha hızlı çalışabilir, ancak kaynak kodları genellikle yorumlanabilir dillere göre daha karmaşık olabilir.

Yorumlanabilir yazılım dilleri genellikle hız açısından derlenmiş dillere göre daha yavaş olabilir, ancak geliştirme sürecinde daha esneklik sağlayabilirler. Yorumlanabilir dilin avantajlarından biri, yazılım geliştirme sürecinde hızlı prototipleme ve hata ayıklama yapma yeteneğidir.

JavaScript'in tarihsel gelişimi:

1995 Netscape Communications Corporation tarafından Brendan Eich liderliğinde geliştirildi ve ilk olarak Netscape Navigator internet tarayıcısında kullanıldı.

1997 JavaScript, ECMA International tarafından standartlaştırıldı ve **ECMAScript** adını aldı. İlk başta **Mocha** adı verilen dil, daha sonra **LiveScript** olarak değiştirildi ve sonunda JavaScript olarak adlandırıldı.

1996 Netscape, JavaScript'i ECMA International'a (European Computer Manufacturers Association) sundu ve bu, ECMAScript standardının doğuşuna yol açtı.

2000'ler JavaScript, **AJAX (Asynchronous JavaScript and XML)** teknolojisinin geliştirilmesi ve **jQuery** gibi kütüphanelerin ortaya çıkmasıyla popülerlik kazandı.

2009 Ryan Dahl tarafından **Node.js** platformu tanıtıldı. Bu, JavaScript'in **sunucu tarafında** da çalışabilmesini sağladı.

2010'lar Angular, **React**, **Vue** gibi JavaScript **frameworkleri** geliştirildi ve yaygınlaştı.

Günümüzde JavaScript, web tarayıcılarından çok daha fazlasında kullanılıyor.

- WEB ⇒ JS, Angular, React, Vue vb.
- Mobil ⇒ React Native, Angular NativeScript vb.
- Desktop ⇒ Electron vb.

Web teknolojilerini (**Node.js** ile birlikte) kullanarak Windows, MacOS ve Linux'ta çalışabilen **çapraz platform destekli uygulamalar** oluşturmanıza olanak tanır. Örneğin, **Visual Studio Code** gibi birçok popüler uygulama Electron kullanılarak geliştirilmiştir.

Javascript Veri Türleri / Tipleri

JS'de veri tipleri **İlkel** ve **Referans/Kompleks** şeklinde 2'ye ayrılır.

Primitive (İlkel) Veriler:

Primitive veriler, basit ve tek bir değeri temsil eder. Bunlar JavaScript'in temel veri tipleridir.

String: Metin verilerini temsil eder. Örnek: "Merhaba, Dünya".

Number: Sayıları temsil eder. Örnek: 42, 3.14.

Boolean: true veya false gibi mantıksal değerleri temsil eder.

Undefined: Değer atanmamış bir değişkeni temsil eder.
Örneğin, let x; ifadesiyle tanımlanan x değişkeni undefined olarak kabul edilir.

Null: Boş bir değeri temsil eder.
Symbol: Benzersiz ve değiştirilemez değerler oluşturmak için kullanılır.

Reference(Referans) veya Complex (Karmaşık) Veriler:

Complex veriler, daha karmaşık yapıları temsil eder ve birden çok değeri içerebilir. Bunlar ilkel verilerin aksine **referans** tipleridir, **yani bellekte bir referansla saklanır ve bu referanslar üzerinden erişilir.**

Object: Nesneler, anahtar-değer çiftlerinden oluşan veri yapılarıdır. Özellikle JSON formatı nesneleri temsil etmek için yaygın olarak kullanılır.

```
let person = {
  name: "John",
  age: 30,
  isStudent: false
};
```

Array: Dizi, sıralı verileri depolamak için kullanılır ve birden çok değeri içerebilir.

```
let numbers = [1, 2, 3, 4, 5];
```

Function: Fonksiyonlar da bir tür karmaşık veridir ve işlevselliği tanımlarlar. JS'de fonksiyonlar birer **nesne olarak kabul edilir.**

```
function merhabaDunya() {
  console.log("Merhaba, Dünya!");
}

merhabaDunya();
```

Date (Tarih): Date türü, tarih ve saat bilgilerini içeren bir nesne oluşturmak için kullanılır.

```
let currentDate = new Date();
```

RegExp (Regular Expression): Düzenli ifadeleri temsil etmek için kullanılır.

```
/* Bir metin içinde belirli bir ifadeyi arama */

let text = "JavaScript çok güçlü bir dil.";
let reg = /güçlü/;
let result = reg.test(text);

console.log(result);  //"güçlü" metin içinde bulunduğu için true veya 1 dönecek.
```

Bir ilkel verinin kopyalanması **sadece değeri kopyalar** ve **orijinal veriyi etkilemez**. Ancak karmaşık veriler referans tipleri olduğundan, bir değişkenin başka bir değişkene atanması sadece **referansı kopyalar**, bu nedenle **orijinal veri üzerinde** değişiklik yapılırsa diğer değişkenler de **etkilenir.**

Değişken Tanımlama Anahtar Kelimeleri

var, let, ve **const** JS'de kullanılan üç farklı değişken tanımlama anahtar kelimeleridir.

var: Eski ecmaScript sürümlerinde daha fazla görebileceğimiz bir anahtar kelimedir.

let: let, ECMAScript 6 ile tanıtılan değişken tanımlama anahtar kelimelerinden biridir.

- **let ile tanımlanan değişkenler blok kapsamına sahiptir. Yani, bir değişken bir bloğun içinde tanımlandığında, sadece bu blok içinde erişilebilir.**
- **let ile aynı isimde bir değişken birden fazla kez aynı kapsamda tanımlanamaz.**

const: ES6 ile tanıtılan bir diğer değişken tanımlama anahtar kelimesidir.

- **const ile tanımlanan değişkenler, let gibi blok kapsamına sahiptir.**
- **let'ten Farklı olarak const ile tanımlanan değişkenlerin değeri bir kez atanır ve daha sonra değiştirilemez (immutable). Yani, bir kez bir değer atandığında, bu değer değiştirilemez.**
- **const ile aynı isimde bir değişken birden fazla kez aynı kapsamda tanımlanamaz.**

Anahtar kelimelerin en temel farkları

Hoisting Davranışı Nedir?

JavaScript'te değişken ve fonksiyon deklarasyonlarının, kodun çalıştırılması öncesinde yukarı taşınması anlamına gelir. Bu, bir değişkenin veya fonksiyonun tanımlanmadan önce kullanılabilmesini sağlar. **Ancak, sadece deklarasyonlar yukarı taşınır, atamalar (değer atamaları) taşınmaz.**

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

```
console.log(x); // ReferenceError: Cannot access 'x' before initialization
let x = 5;
```

var: Deklarasyon ve atama aynı anda hoisted edilir. Bu nedenle, değişken deklare edilmeden önce kullanılabilir, ancak atandığı değeri **undefined** olacaktır.

let ve const: Sadece deklarasyon hoisted edilir. Atama işlemi, deklarasyonun olduğu satırdan önce gerçekleşmez. Bu nedenle, değişken deklare edilmeden önce kullanılmaya çalışıldığında bir **referans hatası (ReferenceError)** alınır.

- var ile tanımlanan değişkenler aynı isimde birden fazla kez tanımlanabilir.
- var ile tanımlanan değişkenler, **işlev kapsamında** (function scope) veya **global kapsamda** (global scope) olabilir.
- var ile tanımlanan değişkenler **hoisting** özelliğine sahiptir, yani deklarasyonları koddaki yukarı taşınır ve tanımlanmadan önce kullanılabilir.

```
var x = 5;
if (true) {
  var x = 10;
  console.log(x); // 10
}
console.log(x); // 10
```

- let ile tanımlanan değişkenler **blok kapsamında** (block scope) olabilir. Yani, tanımlandığı süslü parantez içinde geçerlidir.
- let değişkenleri **hoisting** özelliğine sahiptir, ancak deklarasyondan önce kullanılamazlar.
- Aynı isimdeki let değişkenleri, aynı kapsam içinde bir kez tanımlanabilir. Yani, aynı kapsam içinde aynı isimde ikinci bir let değişkeni tanımlanamaz.

```
let y = 5;
if (true) {
  let y = 10;
  console.log(y); // 10
}
console.log(y); // 5
```

- const ile tanımlanan değişkenler de **blok kapsamında** olabilir.
- const değişkenleri **hoisting** özelliğine sahiptir, ancak deklarasyondan önce kullanılamazlar.
- const ile tanımlanan değişkenlere sadece **bir kez değer atanabilir** ve daha sonra değeri **değiştirilemez (immutable).**

```
const z = 5;
z = 10; // Hata! const değişkeni yeniden atanamaz.
```

Değerlerinizin değişmeyeceği senaryolarda **const** kullanmayı tercih etmelisiniz, değerlerin yanlışlıkla değiştirilmesini önler ve kodunuzu daha güvenli hale getirir.
Ancak, bir değişkenin **değerinin ilerleyen zamanlarda değişmesi** gerekiyorsa veya değişkenin kapsamı sınırlıysa, **let** veya **var** kullanabilirsiniz.

Farkları Tablosu; var, let, const

Özellikler	let	**var**	const
Kapsam	Blok kapsamı	**Fonksiyon kapsamı**	Blok kapsamı
Yeniden Tanımlama	Var	**Var**	Hata verir
Hoisting	Hayır	**Evet (Sadece deklarasyon)**	Hayır
İlk Değer Atama	Opsiyonel	**Undefined**	Zorunlu
Global Obje İlişkisi	Hayır	**Evet (Window objesi ile)**	Hayır

JavaScript'te yorum satırları

iki yöntem vardır: tek satırlı yorumlar ve çok satırlı yorum.

Tek Satırlı Yorumlar
Tek satırlı yorumlar, **//** işareti ile başlar ve tüm satır youm satırı olur.

// Comment satırı.
var x = 5; // comment satırı.

Çok Satırlı Yorumlar
/* ile başlar ve *****/*** ile biter. Arasına yorum yazılır.

/*
Comment
comment
biraz daha comment
*****/***

var y = 10;
/*
Burası Çokomelli
*****/***

JavaScript'te "global değişken"

JS'de global değişken ifadesi diğer scope'lardan da erişilebilen anlamına gelir.
Kısaca;

```
var globalDegisken = "Bu global bir değişkendir."; // Global

function ornek() {
    var yerelDegisken = "Bu bir yerel değişkendir.";  // Fonksiyon içinde bir yerel değişken

    console.log(globalDegisken);// globalDegisken'e fonksiyon içinde erişilebilir

    console.log(yerelDegisken);    // yerelDegisken, sadece bu fonksiyon içinde erişilebilir
}

ornek();

console.log(globalDegisken); // globalDegisken, fonksiyon dışında da erişilebilir

// yerelDegisken'e fonksiyon (burdaki scope) haricinden ulaşılamaz.
```

ReferenceError: yerelDegisken is not defined

Javascript’te this ne anlama gelir

this kelimesi, bir fonksiyonun çağrıldığı **bağlamı (context)** referans alır. this'in değeri, fonksiyonun nasıl çağrıldığına ve fonksiyonun içindeki kod çalıştırıldığında belirlenir.

Global Bağlam

***console.log(this); // window** veya **global** (Node.js'te) anlamına gelmekte.*

Nesne Yöntemi Bağlamı
Bir nesne yöntemi olarak çağrıldığında, `this` o nesneyi referans alır.

```
const object = {
  property: 'Hello',
  myMethod: function() {
    console.log(this.property);
  }
};

object.myMethod(); // "Hello"
```

Yeni Oluşturulan Nesne Bağlamı
Bir yapılandırıcı (constructor) fonksiyonu **new** anahtar kelimesi ile çağrıldığında, **this** yeni oluşturulan nesneyi referans alır.

```
function MyConstructor() {
  this.property = 'Hello';
}

const myInstance = new MyConstructor();
console.log(myInstance.property); // "Hello"
```

Explicit Bağlam
call, apply veya **bind** gibi metotlarla bir fonksiyon çağrıldığında, **this** ilk argüman olarak verilen nesneyi referans alır.

```
function myFunction() {
  console.log(this.property);
}
const myObject = { property: 'Hello' };
myFunction.call(myObject); // "Hello"
```

== ile === farkı nedir? Açıklayınız.

== operatörü, sadece değerlerin eşit olup olmadığını kontrol eder. Değerlerin türüne bakılmaz ve gerekirse dönüşümler gerçekleştirilir.

```
5 == '5';      // true, tür dönüşümü yapılır
true == 1;     // true, tür dönüşümü yapılır
null == undefined;  // true
```

=== operatörü, değerlerin hem değer hem de tür açısından eşit olup olmadığını kontrol eder. Bu nedenle, tür dönüşümü yapılmaz.

```
5 === '5';     // false, tür dönüşümü yapılmaz
true === 1;    // false, tür dönüşümü yapılmaz
null === undefined; // false
```

Arrow fonksiyonları (ok fonksiyonları) ve normal (geleneksel) fonksiyonlar arasındaki farklar

- **Yazım syntex Farkı**

  ```
  const arrowFunction = (param1, param2) => {
    // Fonksiyon gövdesi
  };

  function normalFunction(param1, param2) {
    // Fonksiyon gövdesi
  }
  ```

- **Yeniden Tanımlama Farkı**

  Arrow Fonksiyonlar:
  Arrow fonksiyonları yeniden tanımlanamaz.
  Normal Fonksiyonlar:
  Normal fonksiyonlar yeniden tanımlanabilir.

- **This Tanımı Farkı**

  Arrow Fonksiyonlar:
  Arrow fonksiyonları kendi this bağlamını oluşturmazlar onun yerine **bulundukları kapsamdaki this değerini kullanırlar**.

  Normal Fonksiyonlar:
  Normal fonksiyonlar, **çağrıldıkları bağlama (this değeri) dinamik olarak bağlıdır.**

```
const arrowFunction = () => {
  console.log(this); // Arrow fonksiyon içinde this, tanımlandığı bağlamı gösterir
};

function normalFunction() {
  console.log(this); // Normal fonksiyon içinde this, çağrıldığı bağlamı gösterir
};
```

Switch Bloğu İçinde Hatasız Nasıl Değişken Tanımlanır ?

Hatasız tanımlama yapmak için değişkeni switch dışında değer vermeden tanımlanmalı, case içerisinde değer ataması yapılmalıdır.

```
let durum = "A";
let message;

switch (durum) {
  case "A":
    message = "Durum A seçildi.";
    console.log(message);
    break;

  case "B":
    message = "Durum B seçildi."; // Geçerli: message değişkenini güncelle
    console.log(message);
    break;

  default:
    console.log("Geçersiz durum");
}
```

Pure fonksiyonlar öngörülebilir, test edilebilir ve bağımlılıkları az olan fonksiyonlardır.

Belirli Girdi - Belirli Çıktı: Aynı girdilerle çağrıldığında, bir pure fonksiyon her zaman aynı çıktıyı üretir. Bu özellik, fonksiyonun davranışının tahmin edilebilir ve istikrarlı olmasını sağlar.

Yan Etkisizlik: Bir pure fonksiyon, global ile etkileşime geçmez ve yan etkisi yoktur. Yan etki, fonksiyonun global durumu değiştirmesi anlamına gelir. Pure fonksiyonlar, sadece kendi içsel hesaplamalarına dayanır ve global durumunu değiştirmez.

```
// Pure olmayan fonksiyon yan etki: Her çağrıda farklı çıktı ve global scope durumu değişir

let toplam = 0;
function toplamaYanEtkili(a) {
  toplam += a; // Yan etki: dış dünyada durumu değişir
  return toplam;
}
console.log(toplamaYanEtkili(3)); // 3
console.log(toplamaYanEtkili(5)); // 8
```

NaN nedir ?

"Not a Number" (Sayı Değil) anlamındadır. NaN, **matematiksel** bir işlemin sonucunun sayı olmadığını temsil eder. **NaN değeri, tip olarak bir sayıdır, ancak matematiksel bir işlemle elde edilen geçersiz bir sayıdır.**

```
let sonuc = 0 / 0;  // NaN
console.log(sonuc);

let notANumber1 = parseInt("abc");  // NaN, "abc" bir sayıya dönüştürülemez
let notANumber2 = "hello" * 5;      // NaN, "hello" sayı ile çarpılamaz
console.log(notANumber1, notANumber2);

let notANumber3 = Math.sqrt(-1);    // NaN, negatif bir sayının karekökü yoktur
console.log(notANumber3);

let result = 0 / 0;

if (isNaN(result)) {
  console.log("Sonuç NaN'dir.");
} else {
  console.log("Sonuç bir sayıdır.");
}
```

NaN'in ilginç bir özelliği, **herhangi bir değer ile eşit olmaz, kendisiyle bile.**
NaN'i kontrol etmek için isNaN() gibi özel fonksiyonlar kullanmak önemlidir:

```
console.log(NaN === NaN);  // false
console.log(isNaN(NaN));   // true
```

null ve undefined

- undefined, bir değişkenin değeri atanmamış veya bir nesnenin bir özelliği tanımlanmamışsa varsayılan değeridir.
- Bir değişken tanımlanmış ancak değeri atanmamışsa veya bir fonksiyonun dönüş değeri belirtilmemişse, JavaScript undefined değerini kullanır.

  ```
  let x;
  console.log(x); // undefined
  ```

- null, bir değişkenin veya bir nesnenin bilinçli olarak değeri olmadığını belirtmek için kullanılır.

  ```
  let y = null;
  console.log(y); // null
  ```

Null ve Undefined Farkı

Undefined	Null
undefined, bir değişkenin veya özelliğin **henüz atanmamış** olduğu durumu temsil ederken,	null bilinçli olarak bir değişkenin veya **özelliğin boş olduğunu** ifade eder.
undefined bir **türdür** (undefined),	null ise bir **değerdir** (null).

Bir değişkenin varsayılan değeri undefined'dir, ancak bir değişkenin değeri atanmadığında null kullanılmaz.

== (loose equality) operatörü ile karşılaştırıldığında, undefined ve null **eşittir.**
=== (strict equality) operatörü ile karşılaştırıldığında **eşit değil.**

Rest Operatörü

JavaScript programlama dilinde **"spread"** veya **"rest"** olarak adlandırılır. Dizi veya nesne üzerindeki elemanları kolayca **kopyalamak**, **birleştirmek** veya **toplamak** için kullanılır.

Fonksiyonlarda

```
function toplam(...sayilar) {
  let sonuc = 0;
  for (let sayi of sayilar) {
    sonuc += sayi;
  }
  return sonuc;
}

console.log(toplam(1, 2, 3, 4, 5)); // 15
```

Dizilerde

```
const dizi1 = [1, 2, 3];
const dizi2 = [4, 5, 6];

const birlesikDizi = [...dizi1, ...dizi2];

console.log(birlesikDizi); // [1, 2, 3, 4, 5, 6]
```

Nesnelerde

```
const eskiNesne = { ad: 'John', soyad: 'Doe' };
const yeniNesne = { ...eskiNesne, yas: 30 };

console.log(yeniNesne);
// { ad: 'John', soyad: 'Doe', yas: 30 }
```

Object Destructuring

Bir nesnenin özelliklerini ayrı değişkenlere çıkararak kullanma yöntemine denir.

```
const kullanici = {
  ad: 'John',
  soyad: 'Doe',
  yas: 30,
  email: 'john.doe@example.com'
};
```

Object destructuring kullanarak nesnenin özelliklerini çıkartma

```
const { ad, soyad, yas } = kullanici;
console.log(ad);    // 'John'
console.log(soyad);  // 'Doe'
console.log(yas);    // 30
```

Aynı nesnenin elemanlarını farklı değişken adlarına eşitleyerek destructuring edebiliriz.

```
const { ad: isim, soyad: surname, yas: age } = kullanici;
console.log(isim);    // 'John'
console.log(surname); // 'Doe'
console.log(age);     // 30
```

JAVA vs JavaScript

Özellik	Java	JavaScript
Tür	Statik Tip	Dinamik Tip
Çalışma Ortamı	JVM (Java Virtual Mach.)	Tarayıcı, Sunucu
Sözdizimi ve Dil Özellikleri	Nesne yönelimli	Prototip tabanlı
Asenkron Programlama	Geleneksel olarak	Asenkron Tabanlı

Cookie, LocalStorage, SessionStorage Fark Tablosu

Özellik	Cookie	localStorage	sessionStorage
Veri Saklama Yeri	Tarayıcı ve Sunucu arasında	Tarayıcı'da kalıcı olarak	Tarayıcı oturumu boyunca
Veri Boyutu	Küçük (4 KB limiti)	Büyük (5 MB limiti)	Büyük (5 MB limiti)
Veri Türü	String veya sınırlı tipler	String veya nesneler (key-value çiftleri) String veya nesneler (key-value çiftleri)	String veya nesneler (key-value çiftleri)
Erişim Zamanı	Tarayıcı tarafından okunabilir ve yazılabilir	Tarayıcı tarafından okunabilir ve yazılabilir	Tarayıcı tarafından okunabilir ve yazılabilir
Kullanım Alanı	Oturum kimliği, dil tercihi vb. saklamak için	Kalıcı veri saklamak için	Oturum bazlı veri saklamak için
Kullanım Zorunluluğu	Tarayıcı tarafından desteklenir ve kullanıcı tarafından devre dışı bırakılabilir	Tarayıcı tarafından desteklenir ve kullanıcı tarafından devre dışı bırakılabilir	Tarayıcı tarafından desteklenir ve kullanıcı tarafından devre dışı bırakılabilir
Güvenlik	Sınırlı güvenlik, kullanıcı tarafından silinebilir ve değiştirilebilir	Yüksek güvenlik riski, çünkü tüm tarayıcı sekmesi açık olduğunda bile veri saklanır	Yüksek güvenlik riski, çünkü tüm tarayıcı sekmesi açık olduğunda bile veri saklanır

Asenkron ve Senkron işlem farkı

Senkron (Synchronous)
Senkron bir programlama yaklaşımında, kod satırları **sırayla** çalıştırılır ve bir işlem tamamlanmadan diğerine geçilmez. İşlemler birbirine bağlıdır ve sırasıyla gerçekleşir. Bu durumda bir işlemin tamamlanması beklenmeden **diğer işleme geçilmez**.

Asenkron (Asynchronous)
Asenkron programlama, işlemlerin birbirinden bağımsız olarak çalıştığı bir yaklaşımdır. Bir işlem tamamlanmadan **diğerine geçilebilir** ve işlemler arka planda çalışabilir. Bu, genellikle zaman alan işlemler veya ağ istekleri gibi durumlar için kullanılır.

Promise

Promise (Vaad) JavaScript'te **asenkron programlamayı daha etkili hale getiren bir yapıdır.** Promise, özellikle asenkron işlemleri düzenlemenin ve kontrol etmenin yanı sıra, **callback hell** (geri çağrı cehennemi) olarak adlandırılan **karmaşık ve okunması zor kodları önlemenin bir yolu olarak ortaya çıkmıştır.**

- **Pending (Bekleme)**
  Promise'in başlangıç durumu. Bir işlem henüz tamamlanmamıştır.

- **Fulfilled (Gerçekleşti)**
  Promise, **başarılı** bir şekilde tamamlanmıştır ve **sonuç bir değerle dönmüştür.**

- **Rejected (Reddedildi):**
  Promise, bir **hata** ile karşılaşmıştır ve sonuç bir **hata nesnesiyle dönmüştür.,**

Asenkron İşlemlerin Düzenlenmesi
Promise'ler, asenkron işlemleri düzenlemenin ve sıralamanın daha kolay bir yolunu sağlar. Birden çok asenkron işlemi sırayla veya paralel olarak gerçekleştirmek mümkündür.

Callback Hell'i Engeller
Promise'ler, özellikle birden çok asenkron işlemi **iç içe geçmiş callback** fonksiyonları kullanmadan daha düzenli bir şekilde yönetmeyi sağlar. Bu, kodun daha **okunabilir** ve **bakımı daha kolay** olmasını sağlar.

Hata Yönetimi
Promise'ler, hem **başarıyla** tamamlanan işlemlerin hem de **hata** durumlarının daha etkili bir şekilde ele alınmasını sağlar. **then** ve **catch metodları kullanılarak başarı durumu** ve **hata durumu** belirlenebilir.

```
const myPromise = new Promise((resolve, reject) => {
  const success = true;
  if (success) {
    resolve("Başarılı!");
  } else {
    reject("Hata oluştu!");
  }
});
myPromise
  .then((result) => {
    console.log(result); // Başarılı!
  })
  .catch((error) => {
    console.error(error); // Hata oluştu!
  });
```

Promise'in içindeki asenkron işlem **başarıyla** tamamlanırsa **resolve** çağrılır; aksi takdirde **reject** çağrılır. **then** ve **catch** metodları, **Promise'in durumuna göre işlem yapmanızı sağlar.**

İki elemanlı bir objeyi altı farklı şekilde oluşturma

Obje Literali
const obj1 = { key1: "value1", key2: "value2" };

Object Constructor
const obj2 = new Object();
obj2.key1 = "value1";
obj2.key2 = "value2";

Object.create()
const objTemplate = { key1: null, key2: null };
const obj3 = Object.create(objTemplate);
obj3.key1 = "value1";
obj3.key2 = "value2";

Object.assign()
const obj4 = Object.assign({}, { key1: "value1", key2: "value2" });

Spread Operator
const objTemplate = { key1: "value1", key2: "value2" };
const obj5 = { ...objTemplate };

Json Parse ve Json Stringify
const obj6 = JSON.parse('{"key1": "value1", "key2": "value2"}');

2 elemanlı bir objenin key ve value değerlerinin karakter sayısı ile 2 farklı döngü methodu kullanarak yeni bir obje oluşturma

```
const originalObj = {
  key1: "value123",
  key2: "value4567"
};

const newObjForIn = {};
const newObjForEach = {};

for (let key in originalObj) {
  if (originalObj.hasOwnProperty(key)) {
    newObjForIn[key] = key.length + originalObj[key].length;
  }
}

Object.entries(originalObj).forEach(([key, value]) => {
  newObjForEach[key] = key.length + value.length;
});

console.log("newObjForIn:", newObjForIn);
console.log("newObjForEach:", newObjForEach);
```

Kodlama Soruları:

```
var dolap = ["Shirt", "Pant", "TShirt"];

// 1. Soru: Dolap arrayindeki son elemanı silip consola yazdırın
dolap.pop();
console.log(dolap);

// 2. Soru: Dolap arrayindeki ilk elamanı silip yerine “Hat” elemanını gönderip consola yazdır
dolap.shift();
dolap.unshift("Hat");
console.log(dolap);

// 3. Soru: Dolap değişkeninin array olup olmadığını kontrol edin ve sonucu bir değişkene eşitleyin
var isArray = Array.isArray(dolap);
console.log(isArray);

// 4. Soru: Dolap arrayinde “Pant” elemanın olup olmadığını 3 farklı method ile kontrol edin
var includesMethod = dolap.includes("Pant");
var indexOfMethod = dolap.indexOf("Pant") !== -1;
var findMethod = dolap.find(item => item === "Pant") !== undefined;
console.log(includesMethod, indexOfMethod, findMethod);

// 5. Soru: Dolap arrayindeki elemanların karakter sayısını toplayıp geriye döndürecek fonksiyonu
yazın
function toplamKarakterSayisi(arr) {
  return arr.reduce((total, item) => total + item.length, 0);
}

console.log(toplamKarakterSayisi(dolap));

// 6. Soru: Dolap arrayindki tüm elemanları büyük harfe çevirip yeni bir değişkene 3 farklı yöntemle
atayın
var uppercasedArray1 = dolap.map(item => item.toUpperCase());
var uppercasedArray2 = dolap.join(',').toUpperCase().split(',');
var uppercasedArray3 = dolap.reduce((acc, item) => {
  acc.push(item.toUpperCase());
  return acc;
}, []);

console.log(uppercasedArray1, uppercasedArray2, uppercasedArray3);

// 7. Soru: Dolap arrayini index sayıları key olacak şekilde objeye çeviriniz
var dolapObjesi = {};
dolap.forEach((item, index) => {
  dolapObjesi[index] = item;
});

console.log(dolapObjesi);
```

```
// Slice Orijinal array'i değiştirmez, belirtilen aralıktaki elemanları kopyalar.
// Splice Orijinal array'i değiştirir, belirtilen aralıktaki elemanları çıkarır ve/veya yeni elemanlar ekler.

var orijinalArray = [1, 2, 3, 4, 5];
var slicedArray = orijinalArray.slice(1, 4); // 2, 3, 4
console.log(slicedArray); // [2, 3, 4]
console.log(orijinalArray); // [1, 2, 3, 4, 5]

var splicedArray = orijinalArray.splice(1, 3, 6, 7, 8); // 2, 3, 4
console.log(splicedArray); // [2, 3, 4]
console.log(orijinalArray); // [1, 6, 7, 8, 5]


// Array'deki yinelenen sayıları bulma:
const arr = [1, 2, 3, 4, 5, 6, 7, 7, 8, 6, 10];

const duplicates = arr.filter((value, index, self) => {
  return self.indexOf(value) !== index;
});

console.log("Yinelenen sayılar:", duplicates);

// Array'deki tüm yinelenen sayıları silip yeni bir array oluşturma (2 farklı method ile):
// Method 1
const uniqueArray1 = Array.from(new Set(arr));

// Method 2
const uniqueArray2 = [...new Set(arr)];

console.log("Tekil array (Method 1):", uniqueArray1);
console.log("Tekil array (Method 2):", uniqueArray2);

// Array'deki en yüksek ve en düşük değeri bulma (2 farklı method ile):
// Method 1
const max1 = Math.max(...arr);
const min1 = Math.min(...arr);

// Method 2
const max2 = Math.max.apply(null, arr);
const min2 = Math.min.apply(null, arr);

console.log("En yüksek değer (Method 1):", max1);
console.log("En düşük değer (Method 1):", min1);
console.log("En yüksek değer (Method 2):", max2);
console.log("En düşük değer (Method 2):", min2);

// Örnek 1
function job() {
  return new Promise(function (resolve, reject) {
    reject();
  });
}
```

```
let promise = job();

promise
  .then(function () {
    console.log("Success 1");
  })
  .then(function () {
    console.log("Success 2");
  })
  .then(function () {
    console.log("Success 3");
  })
  .catch(function () {
    console.log("Error 1");
  })
  .then(function () {
    console.log("Success 4");
  });

// Örnek 2
function job(state) {
  return new Promise(function (resolve, reject) {
    if (state) {
      resolve("success");
    } else {
      reject("error");
    }
  });
}

let promise2 = job(true);

promise2
  .then(function (data) {
    console.log(data);
    return job(true);
  })
  .then(function (data) {
    if (data !== "victory") {
      throw "Defeat";
    }
    return job(true);
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.log(error);
    return job(false);
  })
  .then(function (data) {
```

```
  console.log(data);
  return job(true);
})
.catch(function (error) {
  console.log(error);
  return "Error caught";
})
.then(function (data) {
  console.log(data);
  return new Error("test");
})
.then(function (data) {
  console.log("Success:", data.message);
})
.catch(function (data) {
  console.log("Error:", data.message);
});
```

- İlk örnekte reject durumu olduğu için catch bloğu çalışacak ve ardından **'Success 4'** yazdırılacaktır.

- İkinci örnekte ise resolve olduğu için sırasıyla **'success', 'Defeat' (catch bloğu çalışacak), 'Error caught' (sonraki then bloğu çalışacak), 'Success: undefined' (çünkü bir hata döndürüldü) ve en son 'Error: test' (catch bloğu çalışacak) yazdırılacaktır.**