LEARN JAVASCRIPT

BEGINNERS EDITION

Turkish Version

A Complete Beginner's Guide to Learn JavaScript

Suman Kunwar
Süleyman Ekmekci

İçindekiler

Adanmışlık

Bu kitap, dünyamızdaki bilgisayarların ve programlama dillerinin ruhuna saygı ve hayranlıkla adanmıştır.

> "Programlama sanatı, karmaşıklığı organize etme, çokluğun üstesinden gelme ve onun piç kaosundan mümkün olduğunca etkili bir şekilde kaçınma sanatıdır."
>
> - Edsger Dijkstra

Copyright

ASIN: B0C53J11V7

Önsöz

"Learn Javascript:Beginner's Edition", sürekli değişen dijital dünyada önemli bir dil olan Javascript'in kapsamlı bir şekilde keşfini sunar. Temel ve pratiklik odaklı olan bu kitap, Javascript öğrenmek isteyen herkese hitap eder.

Temel konulardan başlayarak, kademeli bir şekilde gelişmiş konulara da ele alır. Örneğin değişkenler, veri tipleri, fonksiyonlar, nesne tabanlı programlama, *closures*, *promises* ve modern *syntax* gibi konuları ele alır. Her bölüm bir önceki bölümün devamı gibi ilerler, böylelikle öğrenenler için sağlam bir temel oluşturur, karmaşık kavramların anlaşılmasını sağlar.

Bu kitabın öne çıkan özelliği, konulara pratik yaklaşımıdır. Okuyucuların öğrendiklerini uygulayabilmeleri ve temel becerilerini geliştirebilmeleri için pratik alıştırmalar, kodlama soruları ve gerçek dünya problemleri sunar. Somut örneklerle etkileşimde bulunarak, okuyucular yaygın web geliştirme sorunlarını çözmek ve JavaScript'in yenilikçi çözümler için potansiyelini ortaya çıkarmak konusunda güven kazanırlar.

Closures ve asenkron programlama gibi karmaşık konular, sezgisel açıklamalar ve pratik örneklerle açıklığa kavuşturulur. Basitlik ve sadeliğe yapılan vurgu, her seviyeden öğrencinin temel kavramları etkili bir şekilde kavramasını ve akılda tutmasını sağlar. Kitap, üç bölüme ayrılmıştır ve ilk 14 bölüm temel kavramlara derinlemesine iner. Takip eden dört bölüm, JavaScript'in web tarayıcı programlaması için kullanımını açıklarken, son iki bölüm çeşitli konuları ele alır ve alıştırmalar sunar. *Miscellaneous* bölümü, JavaScript programlamayla ilgili önemli konuları ve senaryoları keşfederken, alıştırmalarla pratik imkanı sunar.

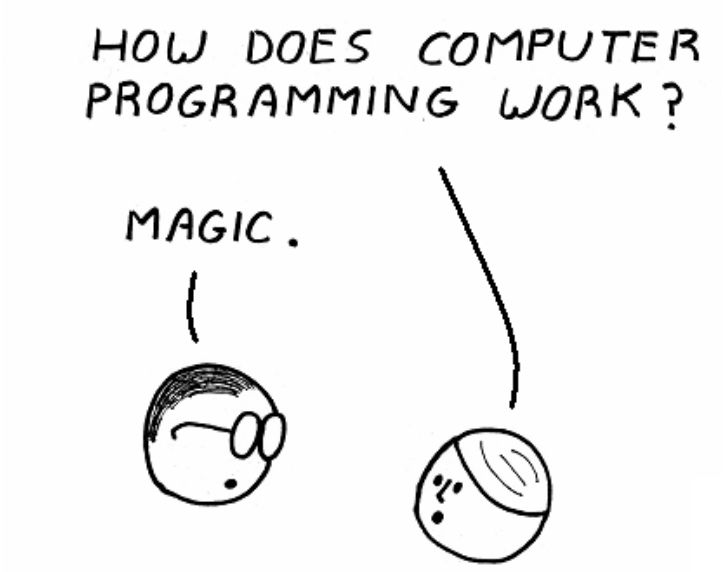
Sonuç olarak, "JavaScript Öğren: Başlangıç Seviyesi", JavaScript'de ustalaşmak ve web geliştirmede başarılı olmak isteyenler için önemli bir kaynaktır. Geniş kapsamı, pratik yaklaşımı, net açıklamaları, gerçek dünya uygulamalarına odaklanması ve sürekli öğrenmeye olan bağlılığı ile bu kitap değerli bir kaynak olarak hizmet vermektedir. Okuyucular, kitabın içeriğine kendilerini kaptırarak dinamik ve etkileşimli web uygulamaları oluşturmak için gerekli bilgi ve becerileri kazanacak ve JavaScript geliştiricileri olarak tüm potansiyellerini ortaya çıkaracaklardır.

Bölüm 1

Giriş

Bilgisayarlar günümüzde çok çeşitli görevleri hızlı ve doğru bir şekilde yerine getirebildikleri için yaygın bir şekilde kullanılmaktadır. İş, sağlık, eğitim, eğlence gibi birçok farklı sektörde kullanılmakta ve birçok insan için günlük yaşamın önemli bir parçası haline gelmiştir. Ayrıca, karmaşık bilimsel ve matematiksel hesaplamalar yapmak, büyük miktarda veriyi depolamak ve işlemek ve dünyanın dört bir yanındaki insanlarla iletişim kurmak için de kullanılırlar.

Programlama, bir bilgisayarın takip etmesi için program adı verilen bir dizi talimat oluşturmayı içerir. Bilgisayarlar verilen görevi tamamlamak için açık ve belirli talimatlara ihtiyaç duyar bu da zaman zaman sıkıcı ve sinir bozucu olabilir.



Programlama dilleri, bilgisayarlara talimat vermek için kullanılan yapay dillerdir. Çoğu programlama görevinde kullanılırlar ve insanların birbirleriyle iletişim kurma şekline dayanırlar. İnsan dilleri gibi, programlama dilleri de yeni kavramları ifade etmek için sözcüklerin ve ifadelerin birleştirilmesine izin verir. İlginç olan, bilgisayarlarla iletişim kurmanın en etkili yolu, insan diline benzeyen bir dil kullanmaktır.

Geçmişte, bilgisayarlarla etkileşim kurmanın temel yolu, BASIC ve DOS komutları gibi dil tabanlı arayüzlerdi. Bunlar büyük ölçüde daha kolay öğrenilebilir olsalar da daha az esneklik sunan görsel arayüzlerle değiştirildi. Ancak *Javascript* gibi bilgisayar dilleri hala kullanımda ve modern web tarayıcılarında ve çoğu cihazda bulunabilir.

JavaScript (*kısaca JS*), web sayfaları, oyunlar, uygulamalar ve hatta sunucular geliştirilirken dinamik etkileşim yaratmak için kullanılan bir programlama dilidir. 1990'larda geliştirilen bir web tarayıcısı olan Netscape'te başlayan JavaScript, bugün en ünlü ve kullanılan programlama dillerinden biridir.

Başlangıçta, web sayfalarını canlandırmak için oluşturulmuş ve yalnızca bir tarayıcıda çalışabilen bir dildi. Şimdi ise, JavaScript motorunu destekleyen herhangi bir cihazda çalışabilir. JavaScript'de Array, Date ve Math gibi standart nesnelerin yanı sıra operatörler, kontrol yapıları ve ifadeler mevcuttur. *İstemci tarafında çalışan Javascript* ve *Sunucu tarafında çalışan Javascript*, ana Javascript'in genişletilmiş versiyonlarıdır.

- *İstemci tarafında çalışan JavaScript* web sayfalarının ve istemci tarayıcılarının geliştirilmesi ve düzenlenmesi için kullanılır. Kullanıcı etkileşimlerine yanıt vermek için: fare tıklamaları, form girişi, sayfa gezinme gibi örnekler verilebilir.
- *Sunucu tarafında çalışan JavaScript* sunuculara, veritabanlarına ve dosya sistemlerine erişim sağlar.

Javascript yorumlanan(*interpreted*) bir dildir. Javascript çalıştırılırken bir yorumlayıcı(*interpreter*) her satırı yorumlar ve çalıştırır. Modern tarayıcı, derleme(*compilation*) için Javascript'i çalıştırılabilir bytecode'a derleyen Just In Time(JIT) teknolojisini kullanır.

> JavaScript'in ilk ismi "LiveScript" idi.

Bu kitap üç ana bölüme ayrılmıştır. İlk 14 bölüm JavaScript dilini anlatmaktadır. Sonraki üç bölüm JavaScript'in web tarayıcılarını programlamak için nasıl kullanıldığını tartışmaktadır. Son iki bölüm ise çeşitli bölümler ve alıştırmalardan oluşmaktadır. JavaScript programlama ile ilgili çeşitli önemli konular ve vakalar Miscellaneous bölümünde anlatılmakta ve bunu alıştırmalar takip etmektedir.

Kod ve onunla ne yapmalı

Kod, bir programı oluşturan yazılı talimatlardır. Bu kitaptaki birçok bölüm çok sayıda kod içerir ve programlamayı öğrenmenin bir parçası olarak kod okumak ve yazmak önemlidir. Örnekleri sadece hızlıca gözden geçirmemelisiniz - dikkatlice okuyun ve anlamaya çalışın. Bu ilk başta zor olabilir, ancak pratik yaptıkça gelişeceksiniz. Aynı şey alıştırmalar için de geçerlidir - onları anladığınızı varsaymadan önce gerçekten bir çözüm yazmaya çalıştığınızdan emin olun. Çözümlerinizi bir JavaScript yorumlayıcısında çalıştırmayı denemeniz de yararlı olacaktır, çünkü bu, kodunuzun doğru çalışıp çalışmadığını görmenizi sağlayacak ve sizi denemeler yapmaya ve alıştırmaların ötesine geçmeye teşvik edebilir.

Tipografik standartlar

Bu kitapta, eşit aralıklı bir yazı tipi kullanılarak yazılmış metinler bir programın unsurlarını temsil eder. Bu yazı tipi, kod örnekleri, program unsurları ve program çıktıları gibi her türlü program öğesinin okuyucular tarafından daha kolay takip edilmesini sağlar.

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

Örneğin, yukarıdaki kod örneği, bir dizi içindeki sayıları birleştirerek bir metin oluşturur ve oluşturulan metni yazdırır. Kod örneğinin sonunda, çıktının nasıl olması gerektiği, iki eğik çizgi işaretiyle (//) "Result" (Sonuç) ibaresinin ardından belirtilmiştir.

```
console.log(txt);

// Result: txt = '45491625'
```

Bu kitapta, bazı kod örneklerinde ve açıklamalarda Türkçe karakterler kullanılsa da, programların dilinin İngilizce olması nedeniyle kod örnekleri tamamen İngilizce'dir. Başarılar! 🍀

Bölüm 2

Temeller

Bu ilk bölümde, programlamanın ve Javascript dilinin temellerini öğreneceğiz.

Programlama kod yazmak demektir. Bir kitap bölümlerden, paragraflardan, cümlelerden, deyimlerden, kelimelerden ve son olarak noktalama işaretleri ve harflerden oluşur, aynı şekilde bir program da daha küçük ve daha küçük bileşenlere ayrılabilir. Şimdilik en önemlisi bir ifadedir. Bir ifade, kitaptaki bir cümleye benzer. Kendi başına bir yapısı ve amacı vardır, ancak etrafındaki diğer ifadelerin bağlamı olmadan o kadar da anlamlı değildir.

Bir ifade yaygın olarak bir *kod satırı* olarak bilinir. Bunun nedeni, ifadelerin tek tek satırlara yazılma eğiliminde olmasıdır. Bu nedenle, programlar yukarıdan aşağıya, soldan sağa doğru okunur. Kodun (kaynak kodu da denir) ne olduğunu merak ediyor olabilirsiniz. Bu, programın tamamını veya en küçük parçasını ifade edebilen geniş bir terimdir. Bu nedenle, bir kod satırı basitçe programınızın bir satırıdır.

İşte basit bir örnek:

```
let hello = "Hello";
let world = "World";

// Message "Hello World"'e eşittir.
let message = hello + " " + world;
```

Bu kod, *interpreter*(*yorumlayıcı*) adı verilen ve kodu okuyup tüm ifadeleri doğru sırada çalıştıran başka bir program tarafından çalıştırılabilir.

Yorumlar

Yorumlar *interpreter* (*yorumlayıcı*) tarafından çalıştırılmayacak ifadelerdir. Yorumlar diğer programcılar için ek açıklamaları veya kodun ne yaptığına dair küçük açıklamaları işaretlemek için kullanılır. Böylece başkalarının kodunuzun ne yaptığını anlamasını kolaylaştırır.

Javascript'te yorumlar 2 farklı şekilde yazılabilir:

- *Tek satırlı yorumlar*:İki eğik çizgi (`//`) ile başlar ve satır sonuna kadar devam eder. Eğik çizgilerden sonraki her şey JavaScript yorumlayıcısı tarafından yok sayılır. Örneğin:

```
// Bu bir yorum satırı, yorumlayıcı tarafından yok sayılacaktır.
let a = "this is a variable defined in a statement";
```

- *Çoklu satırlı yorumlar*: Eğik çizgi ve yıldız işareti (`/*`) ile başlar ve yıldız işareti ve eğik çizgi (`*/`) ile biter. Açılış ve kapanış işaretleri arasındaki her şey JavaScript yorumlayıcısı tarafından yok sayılır. Örneğin:

```
/*
Bu çok satırlı bir yorumdur,
yorumlayıcı tarafından yok sayılacaktır.
*/
let a = "this is a variable defined in a statement";
```

Değişkenler

Programlamayı gerçekten anlamaya yönelik ilk adım cebire geri dönmektir. Eğer okuldan hatırlıyorsanız, cebir aşağıdaki gibi terimlerin yazılmasıyla başlar.

```
3 + 5 = 8
```

Aşağıdaki x gibi bir bilinmeyen girildiğinde hesaplama yapmaya başlarsınız:

```
3 + x = 8
```

x'i yalnız bırakarak bulabilirsiniz.

```
x = 8 - 3
-> x = 5
```

Birden fazla değişken kullandığınızda terimlerinizi daha esnek hale getirmiş olursunuz:

```
x + y = 8
```

x ve y değerlerini değiştirebilirsiniz ve denklem yine de doğru olabilir:

```
x = 4
y = 4
```

veya

```
x = 3
y = 5
```

Aynı durum programlama dilleri için de geçerlidir. Programlamada değişkenler, değerleri tutan kutulardır. Değişkenler çeşitli değerleri ve hesaplamaların sonuçlarını tutabilir. Değişkenler bir `name` (*isim*) ve `value` (*değer*)'e sahiptir. Bu `name` ve `value` `=` ile ayrılır. Farklı programlama dillerinin değişken adı olarak hangi kelimelerin kullanılabileceği konusunda kendi sınırlamaları ve kısıtlamaları olduğunu belirtmekte fayda vardır. Bunun nedeni, belirli kelimelerin dil içindeki bir fonksiyon ya da belirli işlemler için ayrılmış olabilmesidir.

Şimdi JavaScript'te nasıl çalıştığına bakalım. Aşağıdaki kod, iki değişken tanımlar, bu iki değişkenin toplamını hesaplar ve bu sonucu üçüncü bir değişkenin değeri olarak tanımlar.

```
let x = 5;
let y = 6;
let result = x + y;
```

Exercise

20'ye eşit bir `x` değişkeni tanımlayın.

```
let x =
```

ES6 Version

E6 olarak da bilinen ECMAScript 2015 veya ES2015, 2009'dan bu yana JavaScript programlama dilinde yapılan önemli bir güncellemedir. ES6'da değişkenleri tanımlamanın üç yolu vardır.

```
var x = 5;
const y = 'Test';
let z = true;
```

Tanımlama türleri kapsama bağlıdır. `var` anahtar kelimesinin aksine, blok kapsamından bağımsız olarak bir değişkeni global veya yerel olarak tüm bir fonksiyona tanımlar, `let` kapsamları kullandıkları blok ile sınırlı değişkenler tanımlamanızı sağlar. Örneğin:

```
function varTest(){
    var x=1;
    if(true){
        var x=2; // aynı değişken
        console.log(x); //2
    }
    console.log(x); //2
}

function letTest(){
    let x=1;
    if(true){
        let x=2;
        console.log(x); // 2
    }
    console.log(x); // 1
}
```

`const` değişkenleri değişmezdir, yani yeniden atanmalarına izin verilmez.

```
const x = "hi!";
x = "bye"; // buradaki yeniden atama hata verecektir
```

Türler

Bilgisayarlar karmaşıktır ve sayılardan daha karmaşık değişkenler kullanabilirler. İşte bu noktada değişken türleri devreye girer. Değişkenlerin çeşitli türleri vardır ve farklı diller farklı türleri destekler.

En yaygın türler şunlardır:

- **Number**: Sayılar tam sayılar (örneğin, `1`, `-5`, `100`) veya yüzdeli değerler (örneğin, `3.14`, `-2.5`, `0.01`) olabilir. JavaScript tamsayılar ve yüzdeli değerler için ayrı bir türe sahip değildir; her ikisini de sayı olarak ele alır.
- **String**: Stringler, tek tırnak (örneğin, `'hello'`) veya çift tırnak (örneğin, `"world"`) ile temsil edilen karakter dizileridir.
- **Boolean**: Boolean'lar doğru veya yanlış bir değeri temsil eder. Bunlar `true` veya `false` olarak yazılabilir (tırnak işaretleri olmadan).
- **Null**: Null türü, "değer yok" anlamına gelen bir null değerini temsil eder. Tırnak işaretleri olmadan `null` şeklinde yazılabilir.
- **Undefined**: Undefined türü, atanmamış bir değeri temsil eder. Bir değişken tanımlanmış, ancak bir değer atanmamışsa, `undefined` dır.
- **Object**: Obje, her biri bir ad ve değere sahip olan bir özellikler topluluğudur. Küme parantezleri (`{}`) kullanarak bir nesne oluşturabilir ve isim-değer çiftleri kullanarak ona özellikler atayabilirsiniz.
- **Array**: Dizi, bir öğe koleksiyonunu tutabilen özel bir obje türüdür. Köşeli parantez (`[]`) kullanarak bir dizi oluşturabilir ve ona bir değerler atayabilirsiniz.
- **Function**: Fonksiyon, tanımlanabilen ve daha sonra adıyla çağrılabilen bir kod bloğudur. Fonksiyonlar argüman (girdi) kabul edebilir ve bir değer (çıktı) döndürebilir. Bir fonksiyonu `function` anahtar sözcüğünü kullanarak oluşturabilirsiniz.

JavaScript, açık bir şekilde değişkenlerin hangi veri türünü kullandığını belirtmek için ayrıca bir deklarasyon yapmaya gerek duymadan (*loosely typed*) *"gevşek bir şekilde yazılmış"* bir dildir. Sadece bir değişkenin tanımlandığını belirtmek için `var` anahtar kelimesini kullanmanız yeterlidir ve yorumlayıcı, veri türünü bağlama ve alıntıların kullanımına dayanarak belirleyecektir.

Exercise

Aşağıdaki değerlerle üç değişken tanımlayın ve onlara başlangıç değerleri atayın: `age` bir sayı, `name` bir string (dize) ve `isMarried` bir boolean (mantıksal değer) olsun.

```
let age =
let name =
let isMarried =
```

`typeof` operatörü, bir değişkenin veri türlerini kontrol etmek için kullanılır.

```
typeof "John"                 // "string" döndürür
typeof 3.14                   // "number" döndürür
typeof NaN                    // "number" döndürür
typeof false                  // "boolean" döndürür
typeof [1,2,3,4]              // "object" döndürür
typeof {name:'John', age:34}  // "object" döndürür
typeof new Date()             // "object" döndürür
typeof function () {}         // "function" döndürür
typeof myCar                  // "undefined" döndürür
typeof null                   // "object döndürür
```

JavaScript'de kullanılan veri tipleri, içerdikleri değerlere göre iki kategoriye ayrılabilir.

Değer içerebilen veri türleri:

- `string`

- `number`
- `boolean`
- `object`
- `function`

> JavaScript'te kullanılabilen nesne türleri `Object`, `Date`, `Array`, `String`, `Number` ve `Boolean`'dır.

Değer içeremeyen veri türleri:

- `null`
- `undefined`

Primitive (*İlkel*) veri değeri, ek özellikleri ve yöntemleri olmayan basit bir veri değeridir ve bir obje değildir. Primitive veri değerleri değiştirilemezlerdir. 7 Primitive veri türü vardır:

- string
- number
- bigint
- boolean
- undefined
- symbol
- null

Exercise

`person`adında bir değişken tanımlayın ve içinde aşağıdaki özellikleri içeren bir nesneyle başlatalım: `age`(sayı), `name` (dize) ve `isMarried` (boolean).

```
let person =
```

Eşitlik

Bir program yazarken sık sık değişkenlerin diğer değişkenlere göre eşitliğini belirlememiz gerekir. Bu, bir eşitlik operatörü kullanılarak yapılır. En temel eşitlik operatörü `==` operatörüdür. Bu operatör, aynı türden olmasalar bile iki değişkenin eşit olup olmadığını belirlemek kullanılır.

Örneğin, varsayalım:

```
let foo = 42;
let bar = 42;
let baz = "42";
let qux = "life";
```

Bekleneceği gibi `foo == bar` `true` ve `baz == qux` `false` olarak değerlendirilecektir. Ancak, `foo` ve `baz` farklı türler olmasına rağmen `foo == baz` da `true` olarak değerlendirilecektir. Perde arkasında `==` eşitlik operatörü, eşitliklerini belirlemeden önce işlenenlerini aynı türe zorlamaya çalışır. Bu, `===` eşitlik operatörünün tersidir.

Eşitlik operatörü `===` iki değişkenin aynı tipte *ve* aynı değere sahip olmaları durumunda eşit olduğunu belirler. Daha önce olduğu gibi aynı varsayımlarla, bu `foo === bar` hala `true` olarak değerlendirileceği, ancak `foo === baz` artık `false` olarak değerlendirileceği anlamına gelir. `baz === qux` ise yine `false` olarak değerlendirilecektir.

Exercise

`str1` ve `str2` değerlerini karşılaştırmak için `==` ve `===` operatörlerini kullanın.

```
let str1 = "hello";
let str2 = "HELLO";
let bool1 = true;
let bool2 = 1;
// compare using ==
let stringResult1 =
let boolResult1 =
// compare using ===
let stringResult1 =
let boolResult2 =
```

Bölüm 3

Numbers (Sayılar)

JavaScript **sadece tek bir sayı türüne** sahiptir - 64 bit float point. Java'daki `double` ile aynıdır. Diğer programlama dillerinin çoğundan farklı olarak, ayrı bir tamsayı türü yoktur, bu nedenle 1 ve 1.0 aynı değerdir. Bir sayı oluşturmak kolaydır, `var` anahtar sözcüğü kullanılarak diğer değişken türlerinde olduğu gibi yapılabilir.

Sayılar sabit bir değerden oluşturulabilir:

```
// This is a float:
let a = 1.2;

// This is an integer:
let b = 10;
```

Veya başka bir değişkenin değerinden:

```
let a = 2;
let b = a;
```

Tam sayılar 15 basamağa kadar tam olarak doğru bir şekilde temsil edilebilir. Ancak, 16 veya daha fazla basamaklı bir tam sayı tanımlamaya çalıştığınızda, JavaScript bu sayıyı yakınsama (approximation) yöntemiyle temsil etmeye başlar.

```
let x = 999999999999999; // x'in değeri 999999999999999 olacak.
let y = 9999999999999999; // y'nin değeri 10000000000000000 olacak.
```

Sayıların önünde `0x` varsa **hexadecimal** formatta kullanılırlarr.

```
let z = 0xff; // 255
```

Math

`Math` objesi JavaScript'te matematiksel işlemlerin gerçekleştirilmesini sağlar. Statiktir ve bir yapıcısı (**constructor**) yoktur. Math objesi oluşturmadan Math objesinin fonksiyonlarını kullanılabilirsiniz. Bazı matematik özellikleri aşağıda açıklanmıştır:

```
Math.E; // Euler sayısını döndürür
Math.PI; // PI değerini döndürür
Math.SQRT2; // 2'nin karekökünü döndürür
Math.SQRT1_2; // 1/2'nin karekökünü döndürür
Math.LN2; // 2'nin doğal logaritmasını döndürür
Math.LN10; // 10'un doğal logaritmasını döndürür
Math.LOG2E; // E'nin 2 tabanında logaritmasını döndürür
Math.LOG10E; // E'nin 10 tabanında logaritmasını döndürür
```

Bazı matematik yöntemlerine örnek olarak şunlar verilebilir:

```
Math.pow(8, 2); // 64
Math.round(4.6); // 5
Math.ceil(4.9); // 5
Math.floor(4.9); // 4
Math.trunc(4.9); // 4
Math.sign(-4); // -1
Math.sqrt(64); // 8
Math.abs(-4.7); // 4.7
Math.sin((90 * Math.PI) / 180); // 1 (the sine of 90 degrees)
Math.cos((0 * Math.PI) / 180); // 1 (the cos of 0 degrees)
Math.min(0, 150, 30, 20, -8, -200); // -200
Math.max(0, 150, 30, 20, -8, -200); // 150
Math.random(); // 0.44763808380924375
Math.log(2); // 0.6931471805599453
Math.log2(8); // 3
Math.log10(1000); // 3
```

Matematik metoduna erişmek için, gerektiğinde argümanlarla doğrudan metotları çağrılabilir.

Method	Description
`abs(x)`	`x` 'in mutlak değerini döndürür
`acos(x)`	`x` 'in arkkosinüs değerini radyan cinsinden döndürür
`acosh(x)`	`x` 'in hiperbolik arkkosinüsünü döndürür
`asin(x)`	`x` 'in radyan cinsinden arksinüsünü döndürür
`asinh(x)`	`x` 'in hiperbolik arksinüsünü döndürür
`atan(x)`	`x` 'in arktanjantını `-PI/2` ile `PI/2` arasında sayısal bir değer olarak döndürür
`atan2(y,x)`	İki argümanın bölümünün arktanjantını döndürür
`atanh(x)`	`x` 'in hiperbolik arktanjantını döndürür
`crbt(x)`	`x` 'in küp kökünü döndürür
`ceil(x)`	`x` 'in en yakın üst tam sayıya yuvarlanmış hâlini döndürür
`cos(x)`	`x` 'in kosinüsünü radyan cinsinden döndürür
`cosh(x)`	`x` 'in hiperbolik kosinüsünü döndürür
`exp(x)`	`x` 'in üstel fonksiyonunu döndürür
`floor(x)`	`x` 'in en yakın alt tam sayıya yuvarlanmış hâlini döndürür
`log(x)`	`x` 'in doğal logaritmasını döndürür
`max(x,y,z,... n)`	En yüksek değere sahip olan sayıyı döndürür
`min(x,y,z,... n)`	En düşük değere sahip olan sayıyı döndürür
`pow(x,y)`	x'in y. kuvvetini döndürür
random()	0 ile 1 arasında bir sayı döndürür
round(x)	Sayıyı en yakın tam sayıya yuvarlar
sign(x)	x negatifse -1, 0 ise null, pozitifse 1 döndürür
sin(x)	x'in sinüsünü radyan cinsinden döndürür
sinh(x)	x'in hiperbolik sinüsünü döndürür
sqrt(x)	x'in karekökünü döndürür
tan(x)	Bir açının tanjantını döndürür
tanh(x)	x'in hiperbolik tanjantını döndürür
trunc(x)	Bir sayının tam kısmını döndürür

Basic Operators (Temel operatörler)

Sayılar üzerinde matematiksel işlemler, aşağıdaki gibi bazı temel operatörler kullanılarak gerçekleştirilebilir:

- **Toplama operatörü (`+`)**: Toplama operatörü iki sayıyı birbirine ekler. Örneğin:

```
console.log(1 + 2); // 3
console.log(1 + -2); // -1
```

- **Çıkarma operetörü (`-`)**: Çıkarma operatörü bir sayıyı diğerinden çıkarır. Örneğin:

```
console.log(3 - 2); // 1
console.log(3 - -2); // 5
```

- **Çarpma operatörü (`*`)**: Çarpma operatörü iki sayıyı çarpar. Örneğin:

```
console.log(2 * 3); // 6
console.log(2 * -3); // -6
```

- **Bölme operatörü (`/`)**: Bölme operatörü bir sayıyı diğerine böler. Örneğin:

```
console.log(6 / 2); // 3
console.log(6 / -2); // -3
```

- **Remainder (Kalan) operatörü (`%`)**: Kalan operatörü, bir bölme işleminin kalanını döndürür. Örneğin:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

JavaScript yorumlayıcısı soldan sağa doğru çalışır. İfadeleri ayırmak ve gruplamak için matematikte olduğu gibi parantezler kullanılabilir: `c = (a / b) + d`

> JavaScript hem toplama hem de birleştirme için `+` operatörü kullanır. Sayılar eklenirken, dizeler (**string**) birleştirilir.

`NaN` terimi, bir sayının geçerli bir sayı olmadığını belirten ayrılmış bir sözcüktür; bu, sayısal olmayan bir dizeyle aritmetik yaptığımızda ortaya çıkar ve `NaN` (Not a Number) ile sonuçlanır.

```
let x = 100 / "10";
```

`parseInt` fonksiyonu bir değeri bir dize olarak çözümleyip ilk tamsayıyı döndürür.

```
parseInt("10"); // 10
parseInt("10.00"); // 10
parseInt("10.33"); // 10
parseInt("34 45 66"); // 34
parseInt(" 60 "); // 60
parseInt("40 years"); //40
parseInt("He was 40"); //NaN
```

JavaScript'te, mümkün olan en büyük sayının dışında bir sayı hesaplarsak `Infinity` döndürür.

```
let x = 2 / 0; // Infinity
let y = -2 / 0; // -Infinity
```

Exercise

Matematik operatörleri +, -, *, / ve %'yi kullanarak `num1` ve `num2` üzerinde aşağıdaki işlemleri gerçekleştirin.

```
let num1 = 10;
let num2 = 5;

// num1 ile num2'yi toplayın.
let addResult =
// num2'den, num1'i çıkartın.
let subtractResult =
// num1 ile num2'yi çarpın.
let multiplyResult =
// num1 ile num2'yi bölün.
let divideResult =
// Num1'in num2'ye bölümünden kalanı bulun.
let reminderResult =
```

Advanced Operators (İleri Düzey Operatörler)

Operatörler parantez olmadan bir araya getirildiklerinde, uygulanma sıraları operatörlerin *önceliği* tarafından belirlenir. Çarpma `(*)` ve bölme `(/)`, toplama `(+)` ve çıkarmadan `(-)` daha yüksek önceliğe sahiptir.

```
// önce çarpma işlemi, daha sonra toplama işlemi yapılır
let x = 100 + 50 * 3; // 250
// parantez içindeki işlemler önce hesaplanır
let y = (100 + 50) * 3; // 450
// aynı önceliğe sahip işlemler soldan sağa doğru hesaplanır
let z = (100 / 50) * 3;
```

Kod yazarken çeşitli gelişmiş matematik operatörleri kullanılabilir. İşte bazı ileri düzey matematik operatörlerinin bir listesi:

- **Modulo (Kalan) operatörü (`%`)**: Modulo operatörü, bir bölme işleminin kalanını döndürür. Örneğin:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

- **Üs alma operatörü (* *)**: Üs alma operatörü bir sayıyı başka bir sayının kuvvetine yükseltir. Daha yeni bir operatördür ve tüm tarayıcılarda desteklenmez, bu nedenle bunun yerine `Math.pow` işlevini kullanmanız gerekebilir. Örneğin:

```
console.log(2 ** 3); // 8
console.log(3 ** 2); // 9
console.log(4 ** 3); // 64
```

- **Arttırma operatörü (`++`)**: Arttırma operatörü bir sayıyı bir artırır. Kullanılacak elemanın önüne veya sonuna eklenebilir. Örneğin:

```
let x = 1;
x++; // x is now 2
++x; // x is now 3
```

- **Azaltma operatörü (`--`)**: Azaltma operatörü bir sayıyı bir azaltır. Kullanılacak elemanın önüne veya sonuna eklenebilir. Örneğin:

```
let y = 3;
y--; // y is now 2
--y; // y is now 1
```

- **Math objesi**: `Math` objesi, JavaScript'te matematiksel fonksiyonlar ve sabit değerler sağlayan hazır bir objedir. Bir sayının karekökünü bulmak, bir sayının sinüsünü hesaplamak veya rastgele bir sayı oluşturmak gibi ileri düzey matematik işlemleri gerçekleştirmek için `Math` objesinin fonksiyonlarını kullanabilirsiniz. Örneğin:

```
console.log(Math.sqrt(9)); // 3
console.log(Math.sin(0)); // 0
console.log(Math.random()); //0 ile 1 arasında rastgele bir sayı
```

Bunlar JavaScript'te bulunan ileri düzey matematik operatörleri ve fonksiyonlarına sadece birkaç örnek. Program yazarken ileri düzey matematik işlemleri gerçekleştirmek için kullanabileceğiniz çok daha fazlası vardır.

Exercise

`num1` ve `num2` üzerinde işlem yapmak için aşağıdaki ileri düzey operatörleri kullanın.

```
let num1 = 10;
let num2 = 5;

// num1'in değerini artırmak için ++ operatörünü kullanın.
const result1 =
// num2'nin değerini azaltmak için -- operatörünü kullanın.
const result2 =
// num2'yi num1'e eklemek için += operatörünü kullanın.
const result3 =
// num2'yi num1'den çıkarmak için -= operatörünü kullanın.
const result4 =
```

Nullish coalescing operator '??'

`nullish coalescing operator`, `null/undefined` değilse ilk argümanı, `null/undefined` ise ikincisini döndürür. İki soru işareti `??` olarak yazılır. `x ?? y`'nin sonucu şudur:

- eğer `x` tanımlanmışsa, o zaman `x`,
- eğer `y` tanımlanmamışsa, o zaman `y`.

Dile yeni eklenmiştir ve eski tarayıcıların desteklemesi için polyfills'e ihtiyaç duyabilir.

Bölüm 4

Strings (Dizeler)

JavaScript dizeleri, diğer üst düzey dillerden string implementasyonları ile birçok benzerlik paylaşır. Metin tabanlı mesajlar ve verileri temsil ederler. Bu kursta, temelleri ele alacağız. Yeni dizeler nasıl oluşturulur ve bunlar üzerinde ortak işlemler nasıl yapılır.

Bir dizenin örneği:

```
"Hello World";
```

Dizi indeksleri sıfır tabanlı, yani ilk karakterin başlangıç pozisyonu `0` ve diğerleri artımlı sırayla takip eder. Dizi tarafından desteklenen çeşitli yöntemler yeni bir değer döndürür. Bu yöntemler aşağıda açıklanmıştır.

Name	Description
`charAt()`	Belirtilen indeksteki karakteri döndürür
`charCodeAt()`	Belirtilen indeksteki Unicode karakteri döndürür
`concat()`	İki veya daha fazla birleştirilmiş dizeyi döndürür
`constructor`	Dizenin constructor (*yapıcı*) fonksiyonunu döndürür
`endsWith()`	Belirtilen bir değerle biten bir dize olup olmadığını kontrol eder
`fromCharCode()`	Unicode değerlerini karakterler olarak döndürür
`includes()`	Belirtilen bir değerle bir dize içerip içermediğini kontrol eder
`indexOf()`	İlk bulunan elemanın indeksini döndürür
`lastIndexOf()`	Son bulunan indeksini döndürür
`length`	Dizinin uzunluğunu döndürür
`localeCompare()`	Yerel ayar ile iki dize karşılaştırır
`match()`	Bir dize, bir değer veya normal ifade ile eşleştirilir
`prototype`	Nesnenin özelliklerini ve yöntemini eklemek için kullanılır
`repeat()`	Belirtilen sayıda kopya ile yeni bir string döndürür
`replace()`	Değerleri normal bir ifade veya bir string ile bir dize ile değiştirerek döndürür
`search()`	Bir dizenin bir değer veya normal ifade ile eşleşmesine dayalı bir indeks döndürür
`slice()`	Bir dizenin bir kısmını içeren bir string döndürür
`split()`	Bir dizeyi alt dizeler dizisine böler
`startsWith()`	Belirtilen karaktere karşı başlayan dizeleri kontrol eder
`substr()`	Başlangıç indeksinden itibaren dizinin bir kısmını çıkarır
`substring()`	İki indeks arasında dizinin bir kısmını çıkarır
`toLocalLowerCase()`	Host'un yerel ayarı kullanılarak küçük harfler içeren bir string döndürür
`toLocalUpperCase()`	Host'un yerel ayarı kullanılarak büyük harfler içeren bir string döndürür
`toLowerCase()`	Bütün dizeyi küçük harflere döndürülmüş halini döndürür
`toString()`	Dizeyi veya string nesnesini string olarak döndürür
`toUpperCase()`	Bütün dizeyi büyük harflere döndürülmüş halini döndürür
`trim()`	Boşlukların kaldırıldığı bir string döndürür
`trimEnd()`	Sondaki boşlukların kaldırıldığı bir string döndürür
`trimStart()`	Başlangıçtaki boşlukların kaldırıldığı bir string döndürür
`valueOf()`	String veya string nesnesini ilkel değer olarak döndürür

Creation (Oluşturma)

Dizeler, metni tek veya çift tırnak içine alarak tanımlanabilir:

```
// Tek tırnak kullanılabilir
let str = "Our lovely string";

// Çift tırnak da kullanılabilir
let otherStr = "Another nice string";
```

JavaScript'te, dizeler UTF-8 karakterleri içerebilir:

```
"中文 español English हिन्दी العربية português বাংলা русский 日本語 ਪੰਜਾਬੀ 한국어";
```

Ayrıca, `String` yapıcı (*constructor*) fonksiyonu kullanılarak bir dize nesnesi oluşturulabilir:

```
const stringObject = new String("This is a string");
```

Ancak, `String` yapıcı (*constructor*) fonksiyonunu dizeler oluşturmak için kullanmak genellikle önerilmez, çünkü bu, primitive (*ilkel*) dizeler ve dize nesneleri arasında kafa karışıklığına neden olabilir. Dizeler oluşturmak için genellikle string literalleri kullanmak daha iyidir.

Şablon literalleri kullanarak da dizeler oluşturulabilir. Şablon literalleri, ters eğik çizgiler (```) ile sarmalanmış ve değerler için yer tutucular içerebilen dizeler. Yer tutucular, `${}` sözdiziminde gösterilir.

```
const name = "John";
const message = `Hello, ${name}!`;
```

Şablon literalleri birden çok satıra da sahip olabilir ve yer tutucular içinde herhangi bir ifade içerebilir.

> Dizeler çıkarılamaz, çarpılamaz veya bölünemez.

Exercise

`name` ve `age` değerlerini içeren bir dize oluşturmak için bir şablon literali kullanın. Dizenin formatı şu şekilde olmalıdır: "My name is John and I am 25 years old.".

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

// My name is John and I am 25 years old.
let result =
```

Replace

`replace()` metodu, bir karakteri, kelimeyi veya cümleyi başka bir string ile değiştirmemize olanak tanır. Örneğin:

```
let str = "Hello World!";
let new_str = str.replace("Hello", "Hi");

console.log(new_str);

// Sonuç: Hi World!
```

Bir regular expression öğesinin tüm örneklerinde bir değeri `g` değiştiricisi ile değiştirmek için ayarlanır.

Bir değer veya düzenli ifade için bir dize arar ve değer(ler)in değiştirildiği yeni bir dize döndürür. Orijinal dizeyi değiştirmez. Büyük/küçük harfe duyarsız global değiştirme örneğini görelim.

```
let text = "Mr Blue has a blue house and a blue car";
let result = text.replace(/blue/gi, "red");

console.log(result);
//Sonuç: Mr red has a red house and a red car
```

Length (Uzunluk)

JavaScript'te bir dizenin içindeki karakter sayısını `.length` özelliğini kullanarak bilmek kolaydır. `length` özelliği, boşluklar ve özel karakterler dahil olmak üzere dizedeki karakter sayısını döndürür.

```
let size = "Our lovely string".length;
console.log(size);
// size: 17

let emptyStringSize = "".length;
console.log(emptyStringSize);
// emptyStringSize: 0
```

Boş bir dizenin length özelliği `0` 'dır.

> `length` özelliği salt okunur bir özelliktir, bu nedenle ona yeni bir değer atayamazsınız.

Concatenation (Birleştirme)

Birleştirme, iki veya daha fazla dizeyi (*string*) birbirine ekleme işlemidir. Bu, orijinal dizeler içerdiği verileri birleştiren daha büyük bir dize oluşturmak anlamına gelir. Bir dizenin birleştirme, bir dizene bir veya daha fazla dize eklemektir. JavaScript'te bunu aşağıdaki yollarla yapabilirsiniz:

- `+` operatörünü kullanarak
- `concat()` fonksiyonunu kullanarak
- Dizi `join()` fonksiyonunu kullanarak
- Şablon literalini (ES6'da tanıtıldı) kullanarak

Dize `concat()` yöntemi, parametre olarak bir dizi dize alır ve tüm dizeleri birleştirerek yeni bir dize döndürür. Dizi `join()` yöntemi ise, bir dizideki tüm öğeleri tek bir dizeye dönüştürerek ekler.

Şablon literali, ters eğik çizgi (```) kullanır ve çok satırlı dizeler oluşturmanın ve dize ara değişkenlerini kullanmanın kolay bir yoludur. Bir ifade, ters eğik çizginin içinde dolar işareti ve süslü parantezler `{expression}` kullanılarak kullanılabilir.

```
const icon = "👋";
// Şablon Dizeleri kullanarak
`hi ${icon}`;

// join() Metodunu kullanarak
["hi", icon].join(" ");

// concat() Metodunu kullanarak
"".concat("hi ", icon);

// + operatörünü kullanarak
"hi " + icon;
// hi 👋
```

Bölüm 5

Conditional Logic (Koşullu mantık)

Koşul, bir durum için yapılan bir testtir. Koşullar programlama için çeşitli şekillerde çok önemlidir:

Her şeyden önce, koşullar, programınızın işleme koyduğunuz veri ne olursa olsun düzgün çalışmasını sağlamak için kullanılabilir. Eğer veriye körü körüne güvenirseniz, problem yaşarsınız ve programlarınız başarısız olur. Yapmak istediğiniz şeyin mümkün olup olmadığını ve gerekli tüm bilgilerin doğru formatta olup olmadığını test ederseniz, böyle bir şey olmaz ve programınız çok daha tutarlı olur. Bu tür önlemler almak, defansif programlama olarak da bilinir.

Koşulların sizin için yapabileceği bir diğer şey de dallanmaya (*branching*) izin vermektir. Daha önce dallanma diyagramlarıyla karşılaşmış olabilirsiniz, örneğin bir form doldururken. Temel olarak bu, koşulun sağlanıp sağlanmadığına bağlı olarak kodun farklı "dallarının" (bölümlerinin) çalıştırılması anlamına gelir.

Bu bölümde, JavaScript'te koşullu mantığın temelini öğreneceğiz.

If

En basit koşul ifadesi "if" ifadesidir ve sözdizimi (*syntax*) `if (koşul) { bunu yap ... }` şeklindedir. Koşul, süslü parantezlerin içindeki kodun yürütülmesi için doğru (*true*) olmalıdır. Örneğin, aşağıda açıklandığı gibi bir dizeyi test edebilir ve değerine bağlı olarak başka bir dizenin değerini ayarlayabilirsiniz.

```
let country = "France";
let weather;
let food;
let currency;

if (country === "England") {
  weather = "horrible";
  food = "filling";
  currency = "pound sterling";
}

if (country === "France") {
  weather = "nice";
  food = "stunning, but hardly ever vegetarian";
  currency = "funny, small and colourful";
}

if (country === "Germany") {
  weather = "average";
  food = "wurst thing ever";
  currency = "funny, small and colourful";
}

let message =
  "this is " +
  country +
  ", the weather is " +
  weather +
  ", the food is " +
  food +
  " and the " +
  "currency is " +
  currency;

console.log(message);
// 'this is France, the weather is nice, the food is stunning, but hardly ever vegetarian and the currency is funny, small and
```

Koşullar aynı zamanda iç içe de olabilir.

Else

Ayrıca, ilk koşul doğru olmadığında uygulanacak bir `else` ifadesi de vardır. Herhangi bir değere müdahale etmek, ancak özel muamele için özellikle birini ayırmak istiyorsanız bu çok güçlüdür.

```
let umbrellaMandatory;

if (country === "England") {
  umbrellaMandatory = true;
} else {
  umbrellaMandatory = false;
}
```

`else` ifadesi başka bir `if` ile birleştirilebilir. Bir önceki yazıdaki örneği yeniden yapalım.

```
if (country === "England") {
  ...
} else if (country === "France") {
  ...
} else if (country === "Germany") {
  ...
}
```

Exercise

Aşağıdaki değerlerden `num1`in `num2`den büyük olup olmadığını kontrol eden bir koşullu ifade yazın. Eğer büyükse, `result` değişkenine "num1 is greater than num2" atayın. Değilse, "num1 is less than or equal to num2" ifadesini atayın.

```
let num1 = 10;
let num2 = 5;
let result;

// num1'in num2'den büyük olup olmadığını kontrol et
if( condition ) {

}else {

}
```

Switch

`switch` , farklı koşullara göre eylemler gerçekleştiren koşullu bir deyimdir. Koşulları karşılaştırmak için strict (===) karşılaştırmasını kullanır ve eşleşen koşulun kod bloklarını yürütür. `switch` ifadesinin sözdizimi (*syntax*) aşağıda gösterilmiştir.

```
switch (expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
  // code block
}
```

İfade bir kez değerlendirilir ve her bir durumla karşılaştırılır. Eğer eşleşme bulunursa, ilişkili kod bloğu çalıştırılır, aksi halde `default` kod bloğu çalıştırılır. `break` anahtar kelimesi, işlemi durdurur ve istenilen yere yerleştirilebilir. Eğer kullanılmazsa, koşullar eşleşmese bile bir sonraki koşul değerlendirilir.

Aşağıda, switch koşuluna dayalı olarak bir hafta günü adı elde etme örneği verilmiştir.

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
    day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

Birden fazla eşleşme durumunda, **ilk** eşleşen değer seçilir, değilse *default* değer seçilir.Varsayılan değer ve eşleşen bir durum olmadığında, program switch koşullarından sonraki ifadelerle devam eder.

Exercise

Aşağıdaki değerlerden yola çıkarak, dayOfWeek değerini kontrol eden bir `switch` ifadesi yazın. Eğer dayOfWeek "Pazartesi", "Salı", "Çarşamba", "Perşembe" veya "Cuma" ise, result değişkenine "It's a weekday" atanacaktır. Eğer dayOfWeek "Cumartesi" veya "Pazar" ise, result değişkenine "It's the weekend" atanacaktır.

```
let dayOfWeek = "Monday";
let result;
// hafta içi mi yoksa hafta sonu mu olduğunu kontrol et
switch(expression) {
case x:
// code block
break;
case y:
// code block
break;
default:
// code block
}
```

Comparators (Karşılaştırıcılar)

Şimdi koşul kısmına odaklanalım:

```
if (country === "France") {
    ...
}
```

Yukarıdaki kodda koşullu kısım `country` değişkeni ve ardından gelen üç eşittir işaretinden (`===`) oluşuyor. Üç eşit işareti, `country` değişkeninin hem doğru değere (`France`) hem de doğru türe (`String`) sahip olup olmadığını test eder. Koşulları çift eşittir işaretiyle de test edebilirsiniz, ancak `if (x == 5)` gibi bir koşul, hem `var x = 5;` hem de `var x = "5";` için doğru sonucu döndürecektir. Programınızın yaptığı işe bağlı olarak, bu oldukça farklılık yaratabilir. Önerilen yaklaşım, eşitliği her zaman iki eşit işareti (`==` ve `!=`) yerine üç eşit işareti (`===` ve `!==`) ile karşılaştırmanızdır.

Diğer koşullu testler:

- `x > a` : x, a'dan büyük mü?
- `x < a` : x, a'dan küçük mü?
- `x <= a` : x, a'dan küçük veya a'ya eşit mi?
- `x >=a` : x, a'dan büyük veya a'ya eşit mi?
- `x != a` : x, a'ya eşit değil mi?
- `x` : x var mı?

Logical Comparison (Mantıksal Karşılaştırma)

if-else karmaşasından kaçınmak için basit mantıksal karşılaştırmalar kullanılabilir.

```
let topper = marks > 85 ? "YES" : "NO";
```

Yukarıdaki örnekte `?` mantıksal bir operatördür. Kod, marks değeri 85'ten büyükse, yani `marks > 85` ise `topper = YES` ; aksi takdirde `topper = NO` olduğunu söyler. Temel olarak, karşılaştırma koşulunun doğru olduğu kanıtlanırsa, ilk argümana erişilir ve karşılaştırma koşulu yanlışsa, ikinci argümana erişilir. (bkz: Ternary Operator)

Concatenate (Birleştirme)

Bunların yanı sıra, farklı koşulları " `or` " veya " `and` " ifadeleriyle birleştirerek, sırasıyla ifadelerden birinin veya her ikisinin doğru olup olmadığını test edebilirsiniz.

JavaScript'te "or" `||` ve "and" `&&` olarak yazılır.

Diyelim ki x değerinin 10 ile 20 arasında olup olmadığını test etmek istiyorsunuz. Bunu aşağıdaki gibi yapabilirsiniz:

```
if (x > 10 && x < 20) {
    ...
}
```

Eğer o ülkenin "İngiltere" ya da "Almanya" olduğundan emin olmak istiyorsanız:

```
if (country === "England" || country === "Germany") {
    ...
}
```

> **Note**: Tıpkı sayılar üzerindeki işlemler gibi, koşullar da parantez kullanılarak gruplandırılabilir. Örneğin: `if ( (name === "John" || name === "Jennifer") && country === "France")` .

Bölüm 6

Diziler

Diziler, programlamanın temel bir parçasıdır. Bir dizi, verilerden oluşan bir listedir. Bir değişkende bir çok veriyi depolayabiliriz; bu da kodumuzu daha okunabilir ve anlaşılır kılar. Aynı zamanda ilgili veriler üzerinde işlem yapmayı da kolaylaştırır.

Dizilerdeki verilere **eleman** denir.

Basit bir dizi örneği:

```
// 1, 1, 2, 3, 5, ve 8 dizinin elemanlarıdır.
let numbers = [1, 1, 2, 3, 5, 8];
```

Diziler `new` anahtar kelimesiyle veya dizi literalleri ile kolay bir şekilde oluşturulabilir.

```
const cars = ["Saab", "Volvo", "BMW"]; // dizi literalleri kullanarak
const cars = new Array("Saab", "Volvo", "BMW"); // `new` anahtar kelimesi kullanarak
```

Bir dizinin değerlerine erişmek için indeks numarası kullanılır. Bir dizideki ilk elemanın indeksi her zaman "0" ile başlar. İndeks numarası aynı zamanda bir dizideki elemanları değiştirmek için de kullanılır.

```
const cars = ["Saab", "Volvo", "BMW"];
console.log(cars[0]);
// Result: Saab

cars[0] = "Opel"; // dizinin ilk elemanını değiştirme
console.log(cars);
// Result: ['Opel', 'Volvo', 'BMW']
```

> Diziler, özel bir nesne türüdür. Bir dizide nesneler bulunabilir.

Bir dizinin `length` özelliği, elemanlarının sayısını döndürür. Diziler tarafından desteklenen fonksiyonlar aşağıda sıralanmıştır:

Name	Description
`concat()`	İki veya daha fazla birleştirilmiş dizileri döndürür
`join()`	Bir dizideki tüm elemanları bir *string*'e dönüştürüp, birleştirir
`push()`	Bir dizinin sonuna bir veya daha fazla eleman ekler ve uzunluğunu döndürür
`pop()`	Bir dizinin son elemanını kaldırır ve bu elemanı döndürür
`shift()`	Bir dizinin ilk elemanını kaldırır ve bu elemanı döndürür
`unshift()`	Bir dizinin başına bir veya daha fazla eleman ekler ve uzunluğunu döndürür
`slice()`	Bir dizinin bölümünü alır ve yeni bir dizi döndürür
`at()`	Belirtilen indekste eleman varsa elemanı döndürür, yoksa `undefined` döndürür.
`splice()`	Bir diziden elemanları kaldırır, yerlerini değiştirir(opsiyonel) ve diziyi döndürür
`reverse()`	Bir dizinin elemanlarını yerinde tersine çevirir ve dizinin referansını döndürür
`flat()`	Bir dizideki tüm alt dizilerin elemanlarını belirtilen bir derinliğe kadar birleştirerek yeni bir düz diziyi döndürür
`sort()`	Bir dizinin elemanlarını yerinde sıralar ve diziye bir referans döndürür
`indexOf()`	Aranan elemanının ilk eşleştiği indeksini döndürür
`lastIndexOf()`	Aranan elemanının son eşleştiği indeksi döndürür.
`forEach()`	Bir dizinin her elemanında bir *callback* işlevini yürütür ve `undefined` döndürür
`map()`	Her dizi elemanı üzerinde `callback` çalıştırarak döndürülen değeri içeren yeni bir dizi döndürür.
`flatMap()`	Sırasıyla map() ve ardından 1 derinlikte flat() fonksiyonunu çalıştırır
`filter()`	Verilen `callback` fonksiyonunda verilen şartı sağlayan elemanları bir dizi içerisinde döndürür
`find()`	Verilen `callback` fonksiyonunda verilen şartı ilk sağlayan elemanı döndürür.
`findLast()`	Verilen `callback` fonksiyonunda verilen şartı son sağlayan elemanı döndürür.
`findIndex()`	Verilen `callback` fonksiyonunda verilen şartı ilk sağlayan elemanın indeksini döndürür.
`findLastIndex()`	Verilen `callback` fonksiyonunda verilen şartı son sağlayan elemanın indeksini döndürür.
`every()`	Verilen `callback` fonksiyonunda verilen şartı dizinin tüm elemanları sağlıyorsa `true` döndürür.
`some()`	Verilen `callback` fonksiyonunda verilen şartı dizinin en az bir öğesi sağlıyorsa `true` döndürür.
`reduce()`	`reduce()` fonksiyonu, bir dizinin her bir elemanı üzerinde bir işlem yaparak diziyi tek bir değere dönüştürmek için kullanılan bir dizi yöntemidir.
`reduceRight()`	`reduce()` fonksiyonu gibi çalışır fakat son elemandan başlar.

Unshift

`unshift` fonksiyonu, yeni öğeleri sırayla dizi başına ekler. Bu, orijinal diziyi değiştirir ve dizinin yeni uzunluğunu döndürür. Örneğin:

```
let array = [0, 5, 10];
array.unshift(-5); // 4

// Sonuç: array = [-5 , 0, 5, 10];
```

> `unshift()` fonksiyonu, orijinal diziyi değiştirir.

`unshift` fonksiyonu, dizinin başına eklenecek elemanları temsil eden bir veya daha fazla argüman alır. Elemanları verildikleri sırayla ekler, böylece ilk eleman dizinin ilk elemanı olur.

Bir diziye birden fazla eleman eklemek için `unshift` kullanımına bir örnek:

```
const sayilar = [1, 2, 3];
const yeniUzunluk = sayilar.unshift(-1, 0);
console.log(sayilar); // [-1, 0, 1, 2, 3]
console.log(yeniUzunluk); // 5
```

Map

`Array.prototype.map()` fonksiyonu, bir dizi üzerinde yineleme yapar ve *callback function* kullanarak dizi elemanlarını değiştirir. *callback function* daha sonra dizinin her elemanına uygulanır.

`map` kullanımına bir örnek:

```
let newArray = oldArray.map(function (element, index, array) {
  // element: dizi içerisindeki işlenen geçerli öğe
  // index: dizi içinde işlenen geçerli öğenin indeksi
  // array: map methodunun çağrıldığı dizi
  // Yeni diziye eklenecek elemanı döndürün
});
```

Örneğin, sayılardan oluşan bir diziniz var. Siz bu sayı değerlerinin iki katını içeren yeni bir dizi oluşturmak istediğinizi varsayalım. Bunu `map` fonksiyonunu kullanarak şu şekilde yapabilirsiniz:

```
const sayilar = [2, 4, 6, 8];

const ikiKatiSayilar = sayilar.map((sayi) => sayi * 2);

console.log(ikiKatiSayilar);

// Result: [4, 8, 12, 16]
```

Ayrıca `map` fonksiyonunu *arrow function* syntax'i ile de kullanabilirsiniz.

```
let ikiKatiSayilar = sayilar.map((sayi) => {
  return sayi * 2;
});
```

or

```
let ikiKatiSayilar = sayilar.map((sayi) => sayi * 2);
```

`map` fonksiyonu, boş elemanlar için uygulanmaz ve orijinal diziyi değiştirmez.

Spread

Bir dizi veya nesne, Spread(*Yayma*) Operatörü (...) kullanılarak hızlı bir şekilde başka bir dizi veya nesneye kopyalanabilir.

Bazı örnekleri:

```
let dizi1 = [1, 2, 3, 4, 5];

console.log(...dizi1);
// Sonuç: 1 2 3 4 5

let dizi2;
dizi2 = [...dizi1]; //dizi1'i, dizi2'ye kopyalama

console.log(dizi2); //Sonuç: [1, 2, 3, 4, 5]

dizi2 = [6, 7];
dizi1 = [...dizi1, ...dizi2];

console.log(dizi1); //Sonuç: [1, 2, 3, 4, 5, 6, 7]
```

Spread operatörü yalnızca bu özelliği destekleyen modern tarayıcılarda çalışır. Daha eski tarayıcıları desteklemeniz gerekiyorsa, spread operatörü sözdizimini eşdeğer ES5 koduna dönüştürmek için Babel gibi bir *transpiler* kullanmanız gerekecektir.

Shift

`shift` fonksiyonu, bir dizideki ilk elemanı siler ve kalan elemanları sola kaydırır. Orijinal diziyi değiştirir. `shift` fonksiyonunun *syntax*'i:

```
array.shift();
```

Örneğin:

```
let array = [1, 2, 3];
array.shift();

// Sonuç: array = [2,3]
```

Bir dizideki tüm öğeleri kaldırmak için `shift` fonksiyonunu bir döngü ile birlikte de kullanabilirsiniz. Bunu nasıl yapabileceğinize dair bir örnek:

```
while (array.length > 0) {
  array.shift();
}

console.log(array); // Sonuç: []
```

`shift` fonksiyonu yalnızca diziler üzerinde çalışır, argüman nesneleri veya NodeList nesneleri gibi dizilere benzeyen diğer nesneler üzerinde çalışmaz. Bu tür nesnelerden birinin öğelerini kaydırmanız gerekiyorsa, önce `Array.prototype.slice()` fonksiyonunu kullanarak onu bir diziye dönüştürmeniz gerekir.

Pop

`pop()` fonksiyonu, bir diziden son elemanı çıkarır ve bu elemanı döndürür. Bu fonksiyon, dizinin uzunluğunu değiştirir.

`pop()` kullanımı için syntax şu şekildedir:

```
array.pop();
```

Örneğin:

```
let arr = ["one", "two", "three", "four", "five"];
arr.pop();

console.log(arr);

// Sonuç: ['one', 'two', 'three', 'four']
```

Ayrıca `pop` fonksiyonunu bir dizideki tüm öğeleri kaldırmak için bir döngü ile birlikte kullanabilirsiniz. Bunu nasıl yapabileceğinize dair bir örnek:

```
while (array.length > 0) {
  array.pop();
}

console.log(array); // Sonuç: []
```

`pop()` fonksiyonu yalnızca dizilerde çalışır ve diğer dizilere benzer nesnelerde (örneğin, arguments nesneleri veya NodeList nesneleri) çalışmaz. Bu tür nesnelerden elemanları çıkarmak istiyorsanız, önce `Array.prototype.slice()` fonksiyonunu kullanarak onları bir diziye dönüştürmeniz gerekecektir.\

Join

`join` fonksiyonu, dizide bulunan öğeleri tek bir *string*'e dönüştürür. Orijinal diziyi değiştirmez. `join` fonksiyonunun *syntax*'i ise şu şekildedir:

```
array.join([separator]);
```

`seperator` değişkeni isteğe bağlıdır ve dizi elemanlarını hangi karakter ile birbirine bağlayacağını belirtir. Eğer bu değişken verilmezse, dizi elemanları virgül(,) ile ayrılır.

Örneğin:

```
let array = ["bir", "iki", "üç", "dört"];

console.log(array.join(" "));

// Sonuç: bir iki üç dört
```

> Herhangi bir ayırıcı belirtilebilir, ancak varsayılanı virgüldür (,).

Yukarıdaki örnekte boşluk bir ayraç olarak kullanıldı. Ayrıca, dizi benzeri bir nesneyi (arguments nesnesi veya NodeList nesnesi gibi) önce `Array.prototype.slice()` fonksiyonunu kullanarak bir diziye dönüştürmek için de `join` kullanabilirsiniz:

```
function printArguments() {
  console.log(Array.prototype.slice.call(arguments).join(", "));
}

printArguments("a", "b", "c"); // Sonuç: "a, b, c"
```

Length

Dizilerin `length` adında bir özelliği vardır ve adından da anlaşıldığı gibi, dizinin uzunluğunu belirtir.

```
let array = [1, 2, 3];

let l = array.length;

// Sonuç: l = 3
```

`length` özelliği bir dizideki eleman sayısını da düzenler. Örneğin:

```
let fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length = 2;

console.log(fruits);
// Result: ['Banana', 'Orange']
```

Ayrıca `length` özelliğini, bir dizinin son elemanını bulmak için indeks olarak da kullanabilirsiniz. Örneğin:

```
console.log(fruits[fruits.length - 1]); // Result: Orange
```

Bir dizinin sonuna eleman eklemek için de `length` özelliğini kullanabilirsiniz. Örneğin:

```
fruits[fruits.length] = "Pineapple";
console.log(fruits); // Result: ['Banana', 'Orange', 'Pineapple']
```

> Diziye eleman eklendiğinde veya diziden eleman çıkarıldığında `length` özelliği otomatik olarak güncellenir.

Ayrıca `length` özelliğinin bir fonksiyon olmadığını, bu nedenle ona erişirken parantez kullanmanız gerekmediğini de belirtmek gerekir. Bu sadece dizi nesnesinin diğer nesne özellikleri gibi erişebileceğiniz bir özelliğidir.

Push

Bir dizinin sonuna yeni bir eleman eklemek için kullanılır. Bu fonksiyon kullanıldığında dizinin uzunluğunu değiştirir ve yeni uzunluğu döndürür.

`push` kullanımına bir örnek:

```
array.push(element1[, ...[, elementN]]);
```

`element1, ..., elementN` argümanları dizinin sonuna eklenecek elemanları temsil eder.

Örneğin:

```
let array = [1, 2, 3];
array.push(4);

console.log(array);

// Sonuç: array = [1, 2, 3, 4]
```

Ayrıca `push` öğesini, dizi benzeri bir nesnenin (arguments nesnesi veya NodeList nesnesi gibi) sonuna öğe eklemek için, önce `Array.prototype.slice()` fonksiyonunu kullanarak bir diziye dönüştürerek de kullanabilirsiniz:

```
function printArguments() {
  let args = Array.prototype.slice.call(arguments);
  args.push("d", "e", "f");
  console.log(args);
}

printArguments("a", "b", "c"); // Sonuç: ["a", "b", "c", "d", "e", "f"]
```

Şunu unutmamak lazım, `push` fonksiyonu orijinal diziyi değiştirir. Yeni bir dizi oluşturmaz.

For Each

`forEach` fonksiyonu dizideki her eleman için aynı fonksiyonu çalıştırır. Örnek olarak:

```
array.forEach(function (element, index, array) {
  // element: dizide işlenmekte olan eleman
  // index: dizide işlenmekte olan geçerli elemanın indeksi
  // array: forEach'in çağrıldığı dizi
});
```

\ Örneğin, sayılardan oluşan bir dizimiz var ve her sayının iki katını console'a yazdırmak istiyoruz. Bu işlemi `forEach` kullanarak şu şekilde yapabiliriz:

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function (number) {
  console.log(number * 2);
});
```

Ayrıca `forEach` fonksiyonuna *arrow function* ile de fonksiyon verebiliriz.

```
numbers.forEach((number) => {
  console.log(number * 2);
});
```

veya

```
numbers.forEach((number) => console.log(number * 2));
```

`forEach` fonksiyonu orijinal diziyi değiştirmez. Sadece dizinin elemanları üzerinde işlem yapar ve her eleman için sağlanan fonksiyonu çalıştırır.

> `forEach()` fonksiyonu bu şekilde tek başına çağrılmaz.

Sort

`sort` fonksiyonu, bir dizinin öğelerini belirli bir sıraya göre (artan veya azalan) sıralar.

`sort` kullanımı:

```
array.sort([karşılaştırmaFonksiyonu]);
```

`karşılaştırmaFonksiyonu` argümanı isteğe bağlıdır ve sıralama düzenini tanımlayan bir fonksiyon belirtir. Atlanırsa, öğeler *string* gösterimlerine göre artan sırada sıralanır.

Örneğin:

```
let city = ["California", "Barcelona", "Paris", "Kathmandu"];
let sortedCity = city.sort();

console.log(sortedCity);

// Sonuç: ['Barcelona', 'California', 'Kathmandu', 'Paris']
```

> Sayılar sıralandığında yanlış sıralanabilirler. Örneğin, "35", "100"'den daha büyüktür, çünkü "3", "1"'den daha büyüktür.

Sayılardaki sıralama sorununu çözmek için karşılaştırma fonksiyonları kullanılır. Karşılaştırma fonksiyonları sıralama düzenlerini tanımlar. Değişkenlere göre **negatif**, **sıfır** veya **pozitif** bir değer döndürür:

- a, b'den önce sıralanmalı ise negatif bir değer
- a, b'den sonra sıralanmalı ise pozitif bir değer
- a ve b eşitse ve sıralamalarının bir önemi yoksa 0

```
const puanlar = [40, 100, 1, 5, 25, 10];
puanlar.sort((a, b) => {
  return a - b;
});

// Sonuç: [1, 5, 10, 25, 40, 100]
```

`sort()` yöntemi, orijinal diziyi değiştirir.

İndeksler

Diyelim ki veri öğelerinden oluşan bir diziniz var, peki bu dizide bulunan belirli bir öğeye erişmek için ne yaparsınız? İşte burada indeksler devreye giriyor. **İndeks** dizideki bir noktayı ifade eder. İndeksler, mantıksal olarak birer birer ilerler ve çoğu programlama dilinde bir dizinin ilk indeksi 0'dır. Bir dizinin belirli bir öğesine erişmek için `[]` kullanılır.

```
// Meyveleri içeren bir dizi
let meyveler = ["elma", "muz", "ananas", "çilek"];

// `meyveler` dizisinin ikinci elemanını `muz` adında bir değişkene atıyoruz.
// Unutmayın, indeksler 0'dan başlıyor. Bu durumda eğer ikinci elemana yani muza erişmek istiyorsak 1. indeksi seçmeliyiz.
// Sonuç: muz = "muz"
let muz = fruits[1];
```

İndeksleri, dizideki bir elemanın değerini değiştirmek için de kullanabilirsiniz.

```
let harfler = ['a', 'b', 'c', 'd', 'e'];
//  indeksler:  0    1    2    3    4
harfler[4] = 'f';
console.log(harfler); // Sonuç: ['a', 'b', 'c', 'd', 'f']
```

Dizinin sınırları dışında bir indeks kullanarak bir elemana erişmeye veya değiştirmeye çalışırsanız (yani, 0'dan küçük veya dizinin uzunluğuna eşit veya daha büyük bir indeks), `undefined` değeri alırsınız.

```
console.log(array[5]); // Output: undefined
array[5] = 'g';
console.log(array); // Result: ['a', 'b', 'c', 'd', 'f', undefined, 'g']
```

Bölüm 7

Loops

Döngüler, döngüdeki bir değişkenin değişmesiyle gerçekleşen tekrarlayan koşullardır. Aynı kodu her seferinde farklı bir değerle tekrar tekrar çalıştırmak istiyorsanız döngüler kullanışlıdır.

Bunu yapmak yerine:

```
doThing(cars[0]);
doThing(cars[1]);
doThing(cars[2]);
doThing(cars[3]);
doThing(cars[4]);
```

Bunun gibi yapabilirsiniz.

```
for (var i = 0; i < cars.length; i++) {
  doThing(cars[i]);
}
```

For

Bir döngünün en kolay biçimi for ifadesidir. Bunun if deyimine benzer bir sözdizimi vardır, ancak daha fazla seçeneği vardır:

```
for (condition; end condition; change) {
    // do it, do it now
}
```

Bir `for` döngüsü kullanarak aynı kodu on kez nasıl çalıştıracağımızı görelim:

```
for (let i = 0; i < 10; i = i + 1) {
  // bu işlemi 10 kere yap.
}
```

> ***Note***: `i = i + 1` , `i++` olarak yazılabilir.

Bir objenin veya dizinin özellikleri arasında döngü yapmak için `for in` döngüsü de kullanılabilir.

```
for (key in object) {
  // çalıştırılacak kod
}
```

Bir obje ve dizi için `for in` döngüsü örnekleri aşağıda gösterilmiştir:

```
const person = { fname: "John", lname: "Doe", age: 25 };
let info = "";
for (let x in person) {
  info += person[x];
}

// Sonuç: info = "JohnDoe25"

const numbers = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}

// Sonuç: txt = '45491625'
```

`Arrays` , `Strings` , `Maps` , `NodeLists` gibi yinelenebilir objelerin değeri `for of` deyimi kullanılarak döngüye sokulabilir.

```
let language = "JavaScript";
let text = "";
for (let x of language) {
  text += x;
}
```

While

While döngüleri, belirtilen bir koşul doğru olduğu sürece bir kod bloğunu tekrarlı olarak yürütür.

```
while (condition) {
  // koşul doğru olduğu sürece kodu çalıştır
}
```

Örneğin, bu örnekteki döngü, i değişkeni 5'ten küçük olduğu sürece kod bloğunu tekrar tekrar yürütecektir:

```
var i = 0,
  x = "";
while (i < 5) {
  x = x + "The number is " + i;
  i++;
}
```

Koşula her zaman doğru dönen bir ifade vermeye dikkat edin! Çünkü bu sonsuz döngü oluşturacaktır.

Do...While

do...while ifadesi, test koşulu yanlış olarak değerlendirilene kadar belirtilen bir ifadeyi yürüten bir döngü oluşturur. İfade yürütüldükten sonra koşul değerlendirilir. do... while için syntax (*sözdizimi*) şöyledir:

```
do {
  // ifade
} while (expression);
```

Örneğin `do...while` döngüsünü kullanarak 10'dan küçük sayıların nasıl yazdırılacağını görelim:

```
var i = 0;
do {
  document.write(i + " ");
  i++;
} while (i < 10);
```

> ***Note***: `i = i + 1`, `i++` olarak yazılabilir.

Bölüm 8

Functions (Fonksiyonlar)

Fonksiyonlar programlamadaki en güçlü ve temel kavramlardan biridir. Fonksiyonlar matematiksel fonksiyonlar gibi dönüşümler gerçekleştirir, **arguments** adı verilen girdi değerlerini alır ve bir çıktı değeri **döndürür**(*return*)

Fonksiyonlar iki şekilde oluşturulabilir: `function declaration` (*fonksiyon tanımı*) veya `function expression` (*fonksiyon ifadesi*) kullanılarak. Fonksiyon ifadesi kullanıldığında *fonksiyon adı* atlanarak fonksiyon bir `anonymous function` (*anonim fonksiyon*) haline getirilebilir. Fonksiyonlar da değişkenler gibi tanımlanmalıdır. Şimdi `x` adında bir *argüman* kabul eden ve **x'in çiftini** döndüren bir `double` fonksiyonu tanımlayalım:

```
// bir fonksiyon tanımlama örneği
function double(x) {
  return 2 * x;
}
```

> *Note:* yukarıdaki fonksiyona tanımlanmadan önce **başvurulabilir**.

JavaScript'te fonksiyonlar da değerdir; değişkenlerde saklanabilir (tıpkı sayılar, dizeler vb. gibi) ve diğer fonksiyonlara argüman (*parametre*) olarak verilebilirler:

```
// bir fonksiyon ifadesi örneği
let double = function (x) {
  return 2 * x;
};
```

> *Note:* yukarıdaki fonksiyon, diğer değişkenler gibi tanımlanmadan önce **referans gösterilemez**.

Callback, başka bir fonksiyona argüman olarak aktarılan bir fonksiyondur.

Bir arrow fonksiyonu, bazı sınırlamalarla birlikte bazı semantik farklılıklara sahip olan geleneksel fonksiyonlara göre kompakt bir alternatiftir. Bu fonksiyonların `this` , `arguments` ve `super` için kendi bağları yoktur ve constructor(*yapıcı*) olarak kullanılamazlar. Bir arrow fonksiyonu örneği:

```
const double = (x) => 2 * x;
```

Arrow fonksiyonundaki `this` anahtar sözcüğü, arrow fonksiyonunu tanımlayan nesneyi temsil eder.

Higher order (Yüksek dereceli Fonksiyonlar)

Daha yüksek dereceli fonksiyonlar, diğer fonksiyonları kullanabilen fonksiyonlardır. Örneğin, bir high order fonksiyon diğer fonksiyonları argüman olarak alabilir ve/veya geri dönüş değeri olarak bir fonksiyon üretebilir. Bu tür *fancy* (*havalı*) fonksiyonel teknikler JavaScript'te ve Python, Lisp gibi diğer yüksek seviyeli dillerde kullanabileceğiniz güçlü yapılardır.

Şimdi `add_2` ve `double` adında iki basit fonksiyon ve `map` adında daha yüksek dereceli bir fonksiyon oluşturacağız. map `iki argüman kabul edecek`, func `ve` list `ve bir dizi döndürecek. func` (ilk argüman), `list` (ikinci argüman) dizisindeki her bir elemana uygulanacak bir fonksiyon olacaktır.

```
// add_2 ve double fonksiyonlarını tanımlayalım
let add_2 = function (x) {
  return x + 2;
};
let double = function (x) {
  return 2 * x;
};

// map 2 argüman kabul eden harika bir fonksiyondur:
//  func    çağrılacak fonksiyon
//  list    üzerinde func çağrılacak değerler dizisi
let map = function (func, list) {
  let output = []; // output list
  for (idx in list) {
    output.push(func(list[idx]));
  }
  return output;
};

map(add_2, [5, 6, 7]); // => [7, 8, 9]
map(double, [5, 6, 7]); // => [10, 12, 14]
```

Yukarıdaki örnekteki fonksiyonlar basittir. Ancak, diğer fonksiyonlara argüman olarak aktarıldıklarında, daha karmaşık fonksiyonlar oluşturmak için beklenmedik şekillerde bir araya getirilebilirler.

Örneğin, kodumuzda `map(add_2, ...)` ve `map(double, ...)` çağrılarını çok sık kullandığımız için, istediğimiz işlemi içlerinde barındıran iki özel amaçlı liste işleme fonksiyonu oluşturmak istediğimize karar verebiliriz. Fonksiyon oluşturmak için aşağıdaki gibi yapabiliriz:

```
process_add_2 = function (list) {
  return map(add_2, list);
};
process_double = function (list) {
  return map(double, list);
};
process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Şimdi girdi olarak bir fonksiyonu alan ve listedeki her girdiye `func` uygulayan bir fonksiyon döndüren `buildProcessor` adında bir fonksiyon oluşturalım.

```
// a function that generates a list processor that performs
let buildProcessor = function (func) {
  let process_func = function (list) {
    return map(func, list);
  };
  return process_func;
};
// calling buildProcessor returns a function which is called with a list input

// using buildProcessor we could generate the add_2 and double list processors as follows:
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Şimdi başka bir örneğe bakalım. Girdi olarak bir `x` sayısı alan ve argümanını `x` ile çarpan bir fonksiyon döndüren `buildMultiplier` adında bir fonksiyon oluşturalım:

```
let buildMultiplier = function (x) {
  return function (y) {
    return x * y;
  };
};

let double = buildMultiplier(2);
let triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

Bölüm 9

Objects (Objeler)

JavaScript'te objeler **değişkendir**, çünkü referans objeye işaret eden değerleri değiştiririz. Bunun yerine, bir ilkel değeri değiştirirken, referansını değiştiriyoruz, bu da şimdi yeni değere işaret ediyor ve bu nedenle ilkel **değişmezdir**. JavaScript'in ilkel türleri `true` , `false` , `numbers` , `strings` , `null` ve `undefined` 'dir. Herhangi bir diğer değer bir `obje` dir. Objeler, `propertyName` : `propertyValue` çiftleri içerir. JavaScript'te bir `obje` oluşturmanın üç yolu vardır:

1. literal

```
let object = {};
// Evet, sadece bir çift süslü parantez!
```

> ***Not:*** bu **önerilen** yoldur.

2. nesne yönelimli

```
let object = new Object();
```

> ***Not:*** Java'ya çok benzer.

3. ve `object.create` kullanarak

```
let object = Object.create(proto[, propertiesObject]);
```

> ***Not:*** bu, belirtilen prototip nesnesi ve özellikleriyle yeni bir obje oluşturur.

Properties (Obje Özellikleri)

Obje özelliği, `propertyName` : `propertyValue` çiftidir, burada **özellik adı yalnızca bir dize olabilir**. Bir dize değilse, bir dizeye dönüştürülür. Özellikleri bir obje **oluştururken** veya **daha sonra** belirtebilirsiniz. Virgülle ayrılmış sıfır veya daha fazla özellik olabilir.

```
let language = {
  name: "JavaScript",
  isSupportedByBrowsers: true,
  createdIn: 1995,
  author: {
    firstName: "Brendan",
    lastName: "Eich",
  },
  // Yes, objects can be nested!
  getAuthorFullName: function () {
    return this.author.firstName + " " + this.author.lastName;
  },
  // Yes, functions can be values too!
};
```

Aşağıdaki kod, bir özelliğin değerini nasıl elde edeceğinizi gösterir.

```
let variable = language.name;
// variable şimdi "JavaScript" dizesini içeriyor.

variable = language["name"];
// Yukarıdaki satırlar aynı şeyi yapar. Fark, ikincisinin bir özellik adı olarak herhangi bir dize kullanmasına izin vermesidi

variable = language.newProperty;
// variable şimdi undefined, çünkü bu özelliği henüz atamamıştık.
```

Aşağıdaki örnek, yeni bir özellik **ekleme** veya **mevcut olanı değiştirme** işlemini nasıl yapacağınızı gösterir.

```
language.newProperty = "new value";
// Artık objede yeni bir özellik var. Özellik zaten varsa, değeri değiştirilecektir.
language["newProperty"] = "changed value";
// Yine, özellikleri her iki şekilde de erişebilirsiniz. İlki (noktalama notasyonu) önerilir.
```

Mutable (Değişebilirlik)

Objeler ve primitive (*ilkel*) değerler arasındaki fark, **objeleri değiştirebileceğimiz**, ancak primitive (*ilkel*) değerlerin **değiştirilemez** olmasıdır.

Örneğin:

```
let myPrimitive = "first value";
myPrimitive = "another value";
// myPrimitive şimdi başka bir dizeye işaret ediyor.
let myObject = { key: "first value" };
myObject.key = "another value";
// myObject aynı objeye işaret ediyor.
```

Bir objenin özelliklerini nokta veya köşeli ayraç gösterimi kullanarak ekleyebilir, değiştirebilir veya silebilirsiniz.

```
let object = {};
object.foo = "bar"; // Add property 'foo'
object["baz"] = "qux"; // Add property 'baz'
object.foo = "quux"; // Modify property 'foo'
delete object.baz; // Delete property 'baz'
```

> Primitive (*ilkel*) değerler (sayılar ve dizeler gibi) değişmezken, objeler (diziler ve objeler gibi) değişebilirdir.

Reference (Referans)

Objeler **asla kopyalanmaz**. Referansla iletilir. Obje referansı, bir nesneye atıfta bulunan bir değerdir. `new` operatörü veya nesne literal sözdizimi kullanarak bir nesne oluşturduğunuzda, JavaScript bir nesne oluşturur ve ona bir referans atar.

İşte obje literal sözdizimi kullanarak bir nesne oluşturma örneği:

```
var object = {
  foo: "bar";
}
```

İşte `new` operatörü kullanarak bir nesne oluşturma örneği:

```
var object = new Object();
object.foo = "bar";
```

Bir nesne referansını bir değişkene atadığınızda, değişken yalnızca nesneye referans tutar, nesneyi kendisini değil. Bu, nesne referansını başka bir değişkene atarsanız, her iki değişkenin de aynı nesneye işaret edeceği anlamına gelir.

Örneğin:

```
var object1 = {
  foo: "bar",
};

var object2 = object1;

console.log(object1 === object2); // Sonuç: true
```

Yukarıdaki örnekte, `obje1` ve `obje2` hem aynı nesneye referans tutan değişkenlerdir. `===` operatörü referansları karşılaştırmak için kullanılır, nesneleri değil ve her iki değişkenin de aynı nesneye referans tuttuğu için `true` döndürür.

> `Object.assign()` yöntemini kullanarak, var olan bir nesnenin kopyası olan yeni bir nesne oluşturabilirsiniz.

Aşağıda referansla bir nesne örneği verilmiştir.

```
// Bir pizzam olduğunu düşünelim
let myPizza = { slices: 5 };
// Ve onu sizinle paylaştım
let yourPizza = myPizza;
// Ben bir dilim daha yiyorum
myPizza.slices = myPizza.slices - 1;
let numberOfSlicesLeft = yourPizza.slices;
// Şimdi 4 dilimiz var çünkü myPizza ve yourPizza
// aynı pizza nesnesine referans verin.
let a = {},
  b = {},
  c = {};
// a, b ve c her biri
// farklı boş nesne
a = b = c = {};
// a, b ve c hepsi
// aynı boş nesneye referans verin
```

Prototype (Prototip)

Her obje, bir prototip objeye bağlıdır ve bu objeden özellikler miras alır. `{}` nesnel literallerden oluşturulan nesneler, JavaScript ile birlikte gelen bir obje olan `Object.prototype` 'a otomatik olarak bağlanır.

Bir JavaScript yorumlayıcısı (tarayıcınızdaki bir modül), aşağıdaki kodda olduğu gibi, almak istediğiniz bir özelliği bulmaya çalıştığında:

```
let adult = { age: 26 },
  retrievedProperty = adult.age;
// The line above
```

Öncelikle, yorumlayıcı, objenin kendisinin sahip olduğu her özelliği kontrol eder. Örneğin, `adult` 'ın yalnızca bir kendi özelliği vardır — `age` . Ancak bunun dışında, aslında `Object.prototype` 'dan devralınan birkaç özelliği daha vardır.

```
let stringRepresentation = adult.toString();
// the variable has value of '[object Object]'
```

`toString` , devralınan bir Object.prototype özelliğidir. Bir objenin string temsilini döndüren bir fonksiyonun değerine sahiptir. Daha anlamlı bir temsil döndürmek istiyorsanız, bunu geçersiz kılabilirsiniz. Sadece adult nesnesine yeni bir özellik ekleyin.

```
adult.toString = function () {
  return "I'm " + this.age;
};
```

Şimdi `toString` fonksiyonunu çağırırsanız, yorumlayıcı objenin kendisinde yeni özelliği bulacak ve duracaktır.

Yorumlayıcı, nesnenin kendisinden ve daha sonra prototipi boyunca bulacağı ilk özelliği alır.

Varsayılan Object.prototype yerine kendi nesnenizi prototip olarak ayarlamak için `Object.create` 'ı şu şekilde çağırabilirsiniz:

```
let child = Object.create(adult);
/* Bu şekilde nesneler oluşturma, varsayılan Object.prototype'u istediğimiz ile kolayca değiştirmemizi sağlar. Bu durumda, chi

child.age = 8;
/* Daha önce, child'ın kendi age özelliği yoktu ve yorumlayıcı bunu bulmak için child'ın prototipine bakmak zorunda kaldı.
Şimdi, child'ın kendi yaşını ayarladığımızda, yorumlayıcı daha ileri gitmeyecek.
Not: adult'ın yaşı hala 26. */
let stringRepresentation = child.toString();
/* Not: child'ın toString özelliğini geçersiz kılmadık, bu nedenle adult'ın yöntemi çağrılacak. adult'ın toString özelliği yok
```

`child` 'ın prototipi `adult` , `adult` 'ın prototipi ise `Object.prototype` 'dur. Bu prototipler dizisi **prototip zinciri** olarak adlandırılır.

Delete (Silmek)

Bir objeden **bir özelliği** kaldırmak için `delete` özelliği kullanılabilir. Bir özellik silindiğinde, objeden kaldırılır ve erişilemez veya numaralandırılamaz (yani, bir for-in döngüsünde görünmez).

İşte `delete` kullanımı için sözdizimi:

```
delete object.property;
```

Örneğin:

```
let adult = { age: 26 },
  child = Object.create(adult);

child.age = 8;

delete child.age;

/* Remove age property from child, revealing the age of the prototype, because then it is not overriden. */

let prototypeAge = child.age;
// 26, because child does not have its own age property.
```

> `delete` operatörü sadece bir objenin kendi özellikleri üzerinde çalışır, miras alınan özellikler üzerinde çalışmaz. Ayrıca `configurable` niteliği `false` olarak ayarlanmış özellikler üzerinde de çalışmaz.

`delete` operatörü objenin prototip zincirini (**prototype chain**) değiştirmez. Sadece belirtilen özelliği objeden kaldırır ve ayrıca objeyi veya özelliklerini gerçekten yok etmez. Sadece özellikleri erişilemez hale getirir. Bir nesneyi yok etmeniz ve belleğini serbest bırakmanız gerekiyorsa, nesneyi `null` olarak ayarlayabilir veya belleği geri almak için bir çöp toplayıcı kullanabilirsiniz.

Enumeration (Numaralandırma)

Enumeration (Numaralandırma), bir objenin özelliklerini yinelemek ve her özellik için belirli bir eylemi gerçekleştirme işlemine atıfta bulunur. JavaScript'te bir objenin özelliklerini numaralandırmanın birkaç yolu vardır.

Bir objenin özelliklerini numaralandırmanın bir yolu, `for-in` döngüsünü kullanmaktır. `for-in` döngüsü, bir objenin enumerable özelliklerini rastgele bir sırayla yineler ve her özellik için bir blok kodunu çalıştırır.

`for in` ifadesi, bir objedeki tüm özellik adlarını yineleyebilir. Numaralandırma, fonksiyonları ve prototip özelliklerini içerecektir.

```
let fruit = {
    apple: 2,
    orange: 5,
    pear: 1,
  },
  sentence = "I have ",
  quantity;
for (kind in fruit) {
  quantity = fruit[kind];
  sentence += quantity + " " + kind + (quantity === 1 ? "" : "s") + ", ";
}
// The following line removes the trailing comma.
sentence = sentence.substr(0, sentence.length - 2) + ".";
// I have 2 apples, 5 oranges, 1 pear.
```

Bir objenin özelliklerini numaralandırmanın başka bir yolu, `Object.keys()` yöntemini kullanmaktır. Bu yöntem, objenin kendi enumerable özellik adlarının bir diziini döndürür.

Örneğin:

```
let object = {
  foo: "bar",
  baz: "qux",
};

let properties = Object.keys(object);
properties.forEach(function (property) {
  console.log(property + ": " + object[property]);
});

// foo: bar
// baz: qux
```

Bölüm 10

Date and Time (Tarih ve Saat)

`date` nesnesi tarih ve saati depolar ve onu yönetmek için yöntemler sağlar. Tarih nesneleri statiktir ve varsayılan bir tarayıcı zaman dilimini kullanarak tam metinli bir dize olarak tarihi görüntüler. `date` oluşturmak için `new Date()` yapıcısını kullanırız ve aşağıdaki şekillerde oluşturulabilir:

```
new Date()
new Date(date string)
new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)
new Date(milliseconds)
```

> Aylar `0` ile `11` arasında belirtilebilir, daha fazlası bir sonraki yıla akacaktır.

`date` tarafından desteklenen yöntemler ve özellikleri aşağıda açıklanmaktadır:

Name	Description
`constructor`	`Date` nesnesinin prototipini oluşturan işlevi döndürür
`getDate()`	Bir ayın 1-31. gününü döndürür
`getDay()`	Bir haftanın 0-6. gününü döndürür
`getFullYear()`	4 basamaklı yılı döndürür
`getHours()`	0-23 saatini döndürür
`getMilliseconds()`	0-999 milisaniyeyi döndürür
`getMinutes()`	0-59 dakikaları döndürür
`getMonth()`	0-11 ayı döndürür
`getSeconds()`	0-59 saniyeleri döndürür
`getTime()`	1970 Ocak 1'den beri milisaniye cinsinden belirli bir tarihin sayısal değerini döndürür
`getTimezoneOffset()`	Dakika cinsinden zaman dilimi ofsetini döndürür
`getUTCDate()`	Evrensel zamana göre bir ayın 1-31. gününü döndürür
`getUTCDay()`	Evrensel zamana göre 0-6. günü döndürür
`getUTCFullYear()`	Evrensel zamana göre 4 basamaklı yılı döndürür
`getUTCHours()`	Evrensel zamana göre 0-23 saatini döndürür
`getUTCMilliseconds()`	Evrensel zamana göre 0-999 milisaniyeyi döndürür
`getUTCMinutes()`	Evrensel zamana göre 0-59 dakikaları döndürür
`getUTCMonth()`	Evrensel zamana göre 0-11 ayı döndürür
`getUTCSeconds()`	Evrensel zamana göre 0-59 saniyeleri döndürür
`now()`	1970 Ocak 1'den beri milisaniye cinsinden sayısal değeri döndürür
`parse()`	Tarih dizesini ayrıştırır ve 1970 Ocak 1'den beri milisaniye cinsinden sayısal değeri döndürür
`prototype`	Özellikler eklemek için izin verir
`setDate()`	Bir ayın gününü ayarlar
`setFullYear()`	Yılı ayarlar
`setHours()`	Saati ayarlar
`setMilliseconds()`	Milisaniyeleri ayarlar
`setMinutes()`	Dakikaları ayarlar
`setMonth()`	Ayı ayarlar
`setSeconds()`	Saniyeleri ayarlar
`setTime()`	Zamanı ayarlar
`setUTCDate()`	Evrensel zamana göre ayın gününü ayarlar
`setUTCFullYear()`	Evrensel zamana göre yılı ayarlar
`setUTCHours()`	Evrensel zamana göre saati ayarlar
`setUTCMilliseconds()`	Evrensel zamana göre milisaniyeleri ayarlar
`setUTCMinutes()`	Evrensel zamana göre dakikaları ayarlar

Name	Description
`setUTCMonth()`	Evrensel zamana göre ayı ayarlar
`setUTCSeconds()`	Evrensel zamana göre saniyeleri ayarlar
`toDateString()`	İnsan tarafından okunabilir formatta tarihi döndürür
`toISOString()`	Tarihi ISO formatına göre döndürür
`toJSON()`	JSON tarihi olarak biçimlendirilmiş bir dize döndürür
`toLocaleDateString()`	Yerel ayar kurallarına göre tarih içeren bir dize döndürür
`toLocaleTimeString()`	Yerel ayar kurallarına göre zaman içeren bir dize döndürür
`toLocaleString()`	Yerel ayar kurallarına göre tarih ve saat içeren bir dize döndürür
`toString()`	Belirtilen tarihi string temsilini döndürür
`toTimeString()`	*time* kısmını insan tarafından okunabilir bir formatta döndürür
`toUTCString()`	Tarihi evrensel formata göre bir dizeye dönüştürür
`toUTC()`	1970 Ocak 1'den beri UTC formatında milisaniye cinsinden değeri döndürür
`valueOf()`	`Date` 'nin ilkel değerini döndürür

Bölüm 11

JSON

Java**S**cript **O**bject **N**otation (JSON), verileri depolamak ve taşımak için kullanılan metin tabanlı bir formattır. JavaScript nesneleri kolayca JSON'a dönüştürülebilir ve bunun tersi de mümkündür. Örneğin:

```
//  Bir Javascript objesi
let myObj = { name:"Ahmet", age:30, city:"Ankara" };

// JSON'a dönüştürme
let myJSON = JSON.stringify(myObj);
console.log(myJSON);
// Result: '{"name":"Ahmet","age":30,"city":"Ankara"}'

// JSON'dan Javascript objesine dönüştürme
let originalJSON = JSON.parse(myJSON);
console.log(originalJSON);

// Result: {name: 'Ahmet', age: 30, city: 'Ankara'}
```

JSON tarafından desteklenen iki yöntem vardır: `stringify` ve `parse` .

Method	Description
`parse()`	Parçalanmış JSON string'inden JavaScript objesini döndürür.
`stringify()`	JavaScript Objesinden JSON string döndürür.

JSON tarafından desteklenen veri tipleri şunlardır:

- string (dize)
- number (sayı)
- array (dizi)
- boolean (mantıksal değer)
- JSON değerleriyle geçerli olan obje
- null (boş değer)

Ancak `fonksiyon` , `date` (tarih) veya `undefined` (tanımsız) JSON formatında desteklenmez.

Bölüm 12

Error Handling (Hata Yönetimi)

Programlamada hatalar çeşitli nedenlerle meydana gelir; bazıları kod hatalarından, bazıları yanlış girdiden ve diğer öngörülemeyen şeylerden kaynaklanır. Bir hata oluştuğunda, kod durur ve genellikle konsolda görülen bir hata mesajı oluşturur.

try... catch

Kodun çalışmasını durdurmak yerine, kodu durdurmadan hataları yakalamayı sağlayan `try...catch` yapısını kullanabiliriz. `try...catch` yapısının iki ana bloğu vardır; `try` ve ardından `catch` .

```
try {
  // kodun devamı
} catch (err) {
  // hata yönetimi
}
```

İlk olarak `try` bloğundaki kod çalıştırılır. Herhangi bir hatayla karşılaşılmazsa `catch` bloğu atlanır. Bir hata oluşursa, `try` çalışması durdurulur ve kontrol sırası `catch` bloğuna taşınır. Hatanın nedeni `err` değişkeninde yakalanır.

```
try {
  // kodun devamı
  alert("Welcome to Learn JavaScript");
  asdk; // Hata: asdk değişkeni tanımlanmadı
} catch (err) {
  console.log("Error has occurred");
}
```

`try...catch` 'in kullanılabilmesi için kodun çalışabilir ve senkronize olması gerekir.

Özel bir hata vermek için `throw` ifadesi kullanılabilir. Oluşan hatalar tarafından üretilen hata objesinin iki ana özelliği vardır.

- **name**: Hata ismi
- **message**: Hata ile ilgili açıklama mesajı

Eğer bir `error` mesajına ihtiyacınız yoksa catch atlanabilir.

try...catch...finally

`try...catch` 'e `finally` adında bir yapı daha ekleyebiliriz, bu kod her durumda çalışır. Yani hata olmadığında `try` den sonra ve hata durumunda bir `catch` den sonra. `try...catch...finally` için sözdizimi (*syntax*) aşağıda gösterilmiştir.

```
try {
  // kodu çalıştırmayı dene
} catch (err) {
  // hataları denetle
} finally {
  // her durumda çalıştır
}
```

Örnek bir kod örneği:

```
try {
  alert("try");
} catch (err) {
  alert("catch");
} finally {
  alert("finally");
}
```

Yukarıdaki örnekte, önce `try` bloğu çalıştırılır ve ardından herhangi bir hata olmadığı için `finally` bloğu çalıştırılır.

Exercise

Pay ve payda olmak üzere iki argüman alan ve aşağıdaki ayarları kullanarak payın paydaya bölünmesinin sonucunu döndüren bir `divideNumbers()` fonksiyonu yazın.

```
function divideNumbers(numerator, denominator) {
try {
// pay ile paydayı bölmek için `try` kodu
} catch (error) {
// hata mesajı yazdırma
} finally {
// yazdırma işlemi tamamlandı
}
// return result
}
let answer = divideNumbers(10, 2);
```

Bölüm 13

Modüller

Gerçek dünyada, bir program yeni işlevlerin ihtiyaçlarını karşılamak için doğal olarak büyür. Büyüyen kod tabanının yapılandırılması ve koruması ek çalışma gerektirir. Gelecekte ödeyecek olsa da, bunu ihmal etmek daha cazip gelebilir. Gerçekte, uygulamanın karmaşıklığını artırır, çünkü sistem hakkında bütüncül bir anlayış oluşturmak zorunda kalırsınız ve herhangi bir parçayı izole bir şekilde incelemekte zorluk yaşarsınız. İkinci olarak, işlevselliğini kullanabilmek için düğümleri çözmek için daha fazla zaman harcamanız gerekmektedir.

Bu sorunları önlemek için *modüller* mevcuttur. Bir `modül` , hangi kod parçalarına bağlı olduğunu ve diğer modüller için hangi işlevselliği sağladığını belirtir. Başka bir module bağlı olan modüller *dependencies*(bağımlılıklar) olarak adlandırılır. Çeşitli modül kütüphaneleri, kodu modüllerde düzenlemek ve talep üzerine yüklenmek için mevcuttur.

- AMD - başlangıçta require.js tarafından kullanılan en eski modül sistemlerinden biridir.
- CommonJS - Node.js sunucusu için oluşturulan modül sistemi.
- UMD - AMD ve CommonJS ile uyumlu modül sistemi.

Modüller birbirlerini yükleyebilir ve işlevselliği değiş tokuş etmek ve birbirlerinin işlevlerini çağırmak için import ve export özel direktiflerini kullanır.

- `export` - mevcut modülün dışarıdan erişilebilir olmasını istediğiniz işlevleri ve değişkenleri etiketler
- `import` - dışarıdan modülden işlevselliği içe aktarır

Modüllerde `import` ve `export` mekanizmasını görelim. `sayHi.js` dosyasından `sayHi` fonksiyonun dışa aktarıldığını görüyoruz.

```
// 📁 sayHi.js
export const sayHi = (user) => {
  alert(`Hello, ${user}!`);
}
```

`main.js` dosyasında `sayHi` işlevi `import` direktifi yardımıyla içe aktarılır.

```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('Kelvin'); // Hello, Kelvin!
```

Burada import direktifi dosya yolunu içe aktararak modülü yükler ve `sayHi` değişkenine atar. Modüller iki şekilde dışa aktarılabilir: **Named**(Adlandırılmış) ve **Default**(Varsayılan). Ayrıca, *Named* dışa aktarmalar satır içi veya ayrı ayrı atanabilir.

```
// 📁 person.js

// inlined named exports
export const name = "Kelvin";
export const age = 30;

// at once
const name = "Kelvin";
const age = 30;
export {name, age};
```

Bir dosyada yalnızca bir tane `default` dışa aktarım olabilir!

```
// 📁 message.js
const message = (name, age) => {
    return `${name} is ${age} years old.`;
};
export default message;
```

Dışa aktarmaya bağlı olarak, iki şekilde içe aktarabiliriz. *Named*(Adlandırılmış) dışa aktarmalar süslü parantezler kullanılarak oluşturulurken, varsayılan dışa aktarmalar kullanılmaz.

```
import { name, age } from "./person.js"; // named export import
import message from "./message.js"; // default export import
```

Modüller atandığında, döngüsel bağımlılıktan kaçınmalıyız. Döngüsel bağımlılık, bir modülün B'ye, B'nin de doğrudan veya dolaylı olarak A'ya bağımlı olduğu durumdur.

Bölüm 14

Regular Expression (Düzenli İfadeler)

Bir düzenli ifade, bir `RegEx` (Regular Expression) nesnesi olarak oluşturulabilir veya bir öntanımlı değer olarak bir `(/)` (forward slash) karakterleri ile bir desen içine yazılabilir. Düzenli ifade oluşturmak için kullanılan sözdizimleri aşağıda gösterilmiştir.

```
// using regular expression constructor
new RegExp(pattern[, flags]);

// using literals
/pattern/modifiers
```

Literaller kullanarak düzenli ifade oluşturmak için bayraklar isteğe bağlıdır. Yukarıda bahsedilen yöntemi kullanarak aynı düzenli ifadeyi oluşturmak için bir örnek aşağıda gösterilmiştir.

```
let re1 = new RegExp("xyz");
let re2 = /xyz/;
```

Her iki yol da bir regex nesnesi oluşturacak ve aynı yöntemlere ve özelliklere sahip olacaktır. Düzenli ifade oluşturmak için dinamik değerlere ihtiyaç duyabileceğimiz durumlar vardır, bu durumda literaller çalışmaz ve constructor'a gitmemiz gerekir.

> Düzenli ifadenin bir parçası olmak için bir forward slash (/) kullanmak istediğimiz durumlarda, forward slash `(/)` karakterini backslash `(\)` ile kaçırmamız gerekir.

Durumlara duyarlı aramalar yapmak için kullanılan farklı değiştiriciler aşağıda listelenmiştir:

- `g` - global arama (ilk eşleşmeden sonra durmaz, tüm eşleşmeleri bulur)
- `i` - duruma duyarlı arama
- `m` - çok satırlı eşleme

Bir düzenli ifadede, bir dizi karakteri bulmak için *Brackets* (*Köşeli Parantezler*) kullanılır. Bazıları aşağıda listelenmiştir.

- `[abc]` - brackets arasında herhangi bir karakter bul
- `[^abc]` - brackets arasında olmayan bir karakter bul
- `[0-9]` - brackets arasında herhangi bir rakam bul
- `[^0-9]` - brackets arasında olmayan herhangi bir karakter bul (sayılar dışında)
- `(x|y)` - | ile ayrılmış alternatiflerden herhangi birini bul

Düzenli ifadede özel bir anlama sahip olan *Metacharacters* (*özel karakterlerdir*). Bu karakterler aşağıda daha ayrıntılı olarak açıklanmıştır:

Metacharacter	Description
`.`	Bir karakter hariç yeni satır veya bir sonlandırıcı eşleştirir
`\w`	Alfanumerik karakter (`[a-zA-Z0-9_]`) eşleştirir
`\W`	`[^a-zA-Z0-9_]` ile aynı olan bir non word karakteri eşleştirir
`\d`	`[0-9]` ile aynı olan herhangi bir rakam karakterini eşleştirir
`\D`	Sayısal olmayan herhangi bir karakterle eşleşir
`\s`	Bir boşluk karakteriyle eşleştirme (boşluklar, sekmeler vb.)
`\S`	Boşluk olmayan bir karakterle eşleştirme
`\b`	Bir kelimenin başında / sonunda eşleştirir
`\B`	Match but not at the begining / end of a word
`\0`	`NULL` karakterini eşleştirir
`\n`	Yeni bir satır karakterini eşleştirir
`\f`	Form feed karakterini eşleştirir
`\r`	Carriage return karakterini eşleştirir
`\t`	Tab karakterini eşleştirir
`\v`	Tab vertical karakterini eşleştirir
`\xxx`	Bir octal sayı ile belirtilen karakteri eşleştirir (ex: `xxx`)
`\xdd`	Bir hexadecimal sayı ile belirtilen karakteri eşleştirir dd
`\udddd`	Bir hexadecimal sayı ile belirtilen Unicode karakterini eşleştirir dddd

RegEx tarafından desteklenen özellikler ve yöntemler aşağıda listelenmiştir.

Name	Description
`constructor`	RegExp nesnesinin protipini oluşturan fonksiyonu döndürür
`global`	`g` düzenleyicisinin ayarlı olup olmadığını kontrol eder
`ignoreCase`	`i` düzenleyicisinin ayarlanıp ayarlanmadığını kontrol eder
`lastIndex`	Bir sonraki eşleşmenin başlatılacağı dizini belirtir
`multiline`	m düzenleyicisinin ayarlı olup olmadığını kontrol eder
`source`	Dizenin metnini döndürür
`exec()`	Eşleşmeyi test eder ve ilk eşleşmeyi döndürür, eşleşme yoksa `null` döndürür
`test()`	Eşleşmeyi test eder ve `true` veya `false` döndürür
`toString()`	Düzenli ifadenin dize değerini döndürür

Bir `complie()` yöntemi düzenli ifadeyi uyumlu hale getirir ve kullanımdan kaldırılmıştır.

Düzenli ifadelerin (*regular expressions*) bazı örnekleri burada gösterilmektedir.

```
let text = "The best things in life are free";
let result = /e/.exec(text); // looks for a match  of e in a string
// result: e

let helloWorldText = "Hello world!";
// Look for "Hello"
let pattern1 = /Hello/g;
let result1 = pattern1.test(helloWorldText);
// result1: true

let pattern1String = pattern1.toString();
// pattern1String : '/Hello/g'
```

Bölüm 15

Classes (Sınıflar)

Sınıflar, bir nesne oluşturmak için kullanılan şablonlardır. Veriler üzerinde çalışmak için verileri kod ile kapsüller (*encapsulation*). Bir sınıf oluşturmak için `class` anahtar sözcüğü kullanılır. Ve bir sınıfla oluşturulan bir objeyi oluşturmak ve başlatmak için `constructor` (yapıcı) adı verilen belirli bir method kullanılır. Araba sınıfının bir örneği aşağıda gösterilmiştir:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age() {
    let date = new Date();
    return date.getFullYear() - this.year;
  }
}

let myCar = new Car("Toyota", 2021);
console.log(myCar.age()); // 1
```

Sınıf, kullanımından önce tanımlanmalıdır.

Sınıf gövdesinde, yöntemler veya constructors (yapıcılar) tanımlanır ve `strict mode` da çalıştırılır. `strict mode` 'a uymayan sözdizimi hataya neden olur.

Static (Statik)

`static` anahtar kelimesi, bir sınıf için statik yöntemleri veya özellikleri tanımlar. Bu yöntemler ve özellikler, sınıfın içinde çağrılır.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let myCar = new Car("Toyota");

console.log(myCar.hello()); // Hata verir
console.log(Car.hello(myCar));
// Sonuç: Hello Toyota
```

Aynı sınıfın başka bir statik yönteminin statik yöntemine veya özelliğine `this` anahtar sözcüğü kullanılarak erişilebilir;

Inheritance (Kalıtım - Miras)

Kalıtım, bir sınıfın mevcut özelliklerini ve yöntemlerini genişlettiği için kodun yeniden kullanılabilirliği açısından yararlıdır. Bir sınıf kalıtımı oluşturmak için `extends` anahtar sözcüğü kullanılır.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return "I have a " + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ", it is a " + this.model;
  }
}

let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()); // I have a Camry, it is a Toyota.
```

> Üst sınıfın prototipi bir `Object` veya `null` olmalıdır.

`super` yöntemi bir constructor(yapıcının) içinde kullanılır ve üst sınıfa atıfta bulunur. Bu sayede, ana sınıfın özelliklerine ve metodlarına erişilebilir.

Access Modifiers (Erişim Düzenleyicileri)

`public` , `private` ve `protected` , sınıfın dışarıdan erişimini kontrol etmek için kullanılan üç erişim düzenleyicisidir. Varsayılan olarak, sınıfın tüm öğeleri (özellikler, alanlar, yöntemler veya işlevler) sınıf dışından herkese açıktır.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let myCar = new Car("Toyota");
console.log(Car.hello(myCar)); // Hello Toyota
```

`private` öğelerine yalnızca sınıf içinden erişilebilir ve dışarıdan erişilemez. Private `#` ile başlamalıdır.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
  #present(carname) {
    return "I have a " + this.carname;
  }
}
let myCar = new Car("Toyota");
console.log(myCar.#present("Camry")); // Error
console.log(Car.hello(myCar)); // Hello Toyota
```

`protected` öğelerine yalnızca sınıfın içinden ve onu türeterek oluşturan sınıflardan erişilebilir. Bunlar, miras alan sınıfın üst sınıfa da erişim kazanması nedeniyle dahili arayüz için kullanışlıdır. `protected` öğeler `_` ile gösterilir.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  _present() {
    return "I have a " + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this._present() + ", it is a " + this.model;
  }
}
let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()); // I have a Toyota, it is a Camry
```

Bölüm 16

Browser Object Model

Tarayıcı obje modeli (Browser Object Model), tarayıcı penceresi ile etkileşim kurmamızı sağlar. `window` objesi tarayıcının penceresini temsil eder ve tüm tarayıcılar tarafından desteklenir.

`window`, tarayıcı için varsayılan objedir, bu nedenle tüm fonksiyonları doğrudan çağırabiliriz.

```
window.alert("Welcome to Learn JavaScript");

alert("Welcome to Learn JavaScript")
```

Benzer bir şekilde, history, screen, navigator, location gibi diğer özellikleri de window objesinin altında çağırabiliriz.

Window

`window` objesi tarayıcı penceresini temsil eder ve tarayıcılar tarafından desteklenir. Global değişkenler, objeler ve fonksiyonlar da pencere objesinin bir parçasıdır.

Global **değişkenler**, window objesinin **özellikleridir** ve **fonksiyonlar**, window objesinin **metodlarıdır**.

Ekran özelliklerine bir göz atalım. Bu özellikler tarayıcı penceresinin boyutunu belirlemek için kullanılır ve piksel cinsinden ölçülür.

- `window.innerHeight` - tarayıcı penceresinin iç yüksekliği
- `window.innerWidth` - tarayıcı penceresinin iç genişliği

> ***Note***: `window.document` ve `document.location` ifadeleri, (DOM) pencere objesinin bir parçası olduğu için aynı anlama gelir.

Pencere metodlarından birkaç örnek:

- `window.open()` - yeni bir pencere açar
- `window.close()` - mevcut pencereyi kapatır
- `window.moveTo()` - mevcut pencereyi taşır
- `window.resizeTo()` - şu anki pencereyi yeniden boyutlandır

Popup

Popups (Açık pencereler) bilgi göstermek, kullanıcı onayı almak veya ek belgelerden kullanıcı girişi almak için ek bir yoldur. Bir popup yeni bir URL'ye gidebilir ve açılan pencereye bilgi gönderebilir. **Uyarı kutusu** (*Alert box*), **Onay kutusu** (*Confirmation box*) ve **Girdi kutusu** (*Prompt box*) pop-up bilgilerini gösterebileceğimiz global fonksiyonlardır.

1. **alert()**: Kullanıcıya bilgi görüntüler ve devam etmek için bir "**OK**" butonu vardır.

```
alert("Örnek alert mesajı!");
```

2. **confirm()**: Bir şeyi onaylamak veya kabul etmek için bir etkileşim kutusu olarak kullanın. Devam etmek için "**Ok**" ve "**Cancel**" vardır. Kullanıcı "**Ok**" seçeneğine tıklarsa "true", "**Cancel**" seçeneğine tıklarsa "false" döndürür.

```
let txt;
if (confirm("Bir butona basın!")) {
  txt = "OK Butonuna bastınız!";
} else {
  txt = "Cancel butonuna bastınız!";
}
```

3. **prompt()**: "**Ok "** ve "**İptal "** düğmeleri ile kullanıcı girdi alır. Kullanıcı herhangi bir girdi değeri sağlamazsa `null` döndürür.

```
//syntax
//window.prompt("sometext","defaultText");

let person = prompt("Lütfen adınızı girin", "Harry Potter");

if (person == null || person == "") {
  txt = "Kullanıcı prompt girmedi";
} else {
  txt = "Merhaba " + person + "! Bugün nasılsınız?";
}
```

Screen

`screen` objesi, geçerli pencerenin oluşturulduğu ekran hakkında bilgi içerir. `screen` objesine erişmek için `window` objesinin `screen` özelliğini kullanabiliriz.

```
window.screen
//veya
screen
```

`window.screen` objesinin farklı özellikleri vardır, bunlardan bazıları aşağıda listelenmiştir:

Property	Description
`height`	Ekranın piksel yüksekliğini temsil eder.
`left`	Mevcut ekranın sol tarafının piksel mesafesini temsil eder.
`pixelDepth`	Ekranın bit derinliğini döndüren salt okunur bir özelliktir.
`top`	Mevcut ekranın üst kısmının piksel mesafesini temsil eder.
`width`	Ekranın piksel genişliğini temsil eder.
`orientation`	Ekran Oryantasyon API'sinde (*Screen Orientation API*) belirtildiği şekilde ekran oryantasyonunu döndürür
`availTop`	Sistem öğeleri tarafından alınmayan üstteki ilk pikseli döndüren salt okunur bir özelliktir.
`availWidth`	Sistem öğeleri hariç ekranın piksel genişliğini döndüren salt okunur bir özelliktir.
`colorDepth`	Renkleri temsil etmek için kullanılan bit sayısını döndüren salt okunur bir özelliktir.

Navigator

`window.navigator` veya `navigator` özelliği salt okunur bir özelliktir ve tarayıcıyla ilgili çeşitli yöntem ve fonksiyonları içerir.

Birkaç örneğine bakalım:

1. **navigator.appName**: Tarayıcı uygulamasının adını verir

```
navigator.appName;
// "Netscape"
```

> ***Note:*** "Netscape" IE11, Chrome, Firefox ve Safari için uygulama adıdır.

2. **navigator.cookieEnabled**: Tarayıcıdaki çerez değerine bağlı olarak bir boolean değeri döndürür.

```
navigator.cookieEnabled;
//true
```

3. **navigator.platform**: İşletim sistemi hakkında bilgi sağlar.

```
navigator.platform;
"MacIntel"
```

Çerezler 🍪

Çerezler, bir bilgisayarda depolanan ve tarayıcı tarafından erişilebilen bilgi parçacıklarıdır.

Web tarayıcısı ile sunucu arasındaki iletişim durumsuzdur, kendisinden önceki isteklerin bilgisini içermez yani her isteği bağımsız olarak işler. Kullanıcı bilgilerini depolamamız ve bu bilgileri tarayıcıya kullanılabilir hale getirmemiz gereken durumlar vardır. Bu gibi durumlarda çerezler aracılığıyla bilgi, gerektiğinde bilgisayardan alınabilir. Çerezler `name-value` (isim-değer) çifti olarak kaydedilir.

```
book = Learn Javascript
```

`document.cookie` özelliği, çerezleri oluşturmak, okumak ve silmek için kullanılır. Çerez oluşturmak oldukça kolaydır, sadece ismi ve değeri sağlamanız gerekmektedir.

```
document.cookie = "book=Learn Javacript";
```

Varsayılan olarak, bir tarayıcı kapatıldığında bir çerez silinir. Kalıcı olmasını sağlamak için son kullanma tarihini (UTC zaman diliminde) belirtmemiz gerekmektedir.

```
document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC";
```

Çerezin hangi sayfa yoluna (*path*) ait olduğunu söylemek için bir parametre ekleyebiliriz. Varsayılan olarak, çerez şu anki sayfaya aittir.

```
document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
```

İşte basit bir çerez örneği.

```
let cookies = document.cookie;
// tüm çerezlere ulaşmanın basit bir yolu.

document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
// çerez oluşturma
```

Geçmiş

Bir web tarayıcısını açtığımızda ve bir web sayfasında gezindiğimizde, geçmiş belleğinde yeni bir girdi oluşturur. Farklı sayfalarda gezinmeye devam ettikçe yeni girdiler geçmiş belleğine eklenir.

Geçmiş objesine erişmek için şunları kullanabiliriz:

```
window.history
// veya
history
```

Farklı geçmiş verileri arasında gezinmek için **history** objesinin `go()` , `foward()` ve `back()` fonksiyonlarını kullanabiliriz.

1. **go()**: Geçmiş belleğinin belirli URL'sinde gezinmek için kullanılır.

   ```
   history.go(-1); // sayfayı geri taşır
   history.go(0);  // şu anki sayfayı yeniler
   history.go(); // şu anki sayfayı yeniler
   history.go(1) // sayfayı ileri taşır
   ```

   > ***Note:*** geçmiş belleğindeki geçerli sayfa konumu **0**'dır.

2. **back()** : Geriye gitmek için `back()` fonksiyonu kullanılır.

   ```
   history.back();
   ```

3. **forward()**: Tarayıcı geçmişindeki bir sonraki listeyi yükler. Tarayıcıdaki ileri düğmesine tıklamaya benzer.

   ```
   history.forward();
   ```

Location

`location` objesi, belgenin geçerli (URL) konumunu almak için kullanılır ve konumunu manipüle etmek için farklı yöntemler sağlar. Geçerli konuma şu şekilde erişilebilir

```
window.location
//veya
document.location
//veya
location
```

> ***Note***: `window.location` ve `document.location` aynı konum objesine referans verir.

Aşağıdaki URL örneğini ele alalım ve `location` objesinin farklı özelliklerini inceleyelim

`http://localhost:3000/js/index.html?type=listing&page=2#title`

```
location.href //mevcut URL'yi yazdırır.
location.protocol //protokolleri yazdırır örn: http,https
location.host //hostname'i port ile birlikte yazdırır örn: localhost:3000
location.hostname //hostname'i yazdırır örn: localhost, www.example.com
location.port //port numarasını yazdırır örn: 3000
location.pathname //sayfa yolunu(*path*) yazdırır örn: /js/index.html
location.search //sorgu dizelerini yazdırır örn: ?type=listing&page=2
location.hash //title gibi bileşen tanımlayıcısını yazdırır
```

Bölüm 17

Events (Olaylar)

Programlamada, events (*olaylar*), bir sistem tarafından size bildirilen ve bunlara yanıt verebilmeniz için sistemdeki eylemler veya olaylardır. Örneğin, reset düğmesine tıkladığınızda girişi temizler.

Klavyeden gelen etkileşimler, anahtarın durumunu serbest bırakılmadan önce yakalamak için sürekli olarak okunmalıdır. Diğer zaman alıcı hesaplamalar yapmak, bir klavye basımını kaçırmanıza neden olabilir. Bu, bazı ilkel makinelerin giriş işleme mekanizması used to. Bir adım daha ileri gitmek, bir kuyruk kullanmaktır, yani yeni olaylar için kuyruğu periyodik olarak kontrol eden ve buna tepki veren bir programdır. Bu yaklaşıma *polling* denir.

Bu yaklaşımın ana dezavantajı, her an kuyruğa bakması gerektiğinden, bir olay tetiklendiğinde kesintilere neden olmasıdır. Bunun için daha iyi bir mekanizma, kod bir olay gerçekleştiğinde bilgilendirmektir. Modern tarayıcılar bunu, belirli olaylar için işlevleri *handlers* olarak kaydetmemize izin vererek yaparlar.

```
<p>Click me to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("clicked");
  });
</script>
```

Burada, `addEventListener` `window` nesnesi (tarayıcı tarafından sağlanan yerleşik nesne) üzerinde çağrılır ve tüm `window` için bir handler kaydeder. `addEventListener` yöntemini çağırmak, ilk argümanında tanımlanan olay meydana geldiğinde ikinci argümanının çağrılmasını kaydeder.

> Olay dinleyicileri, yalnızca olay, kaydolundukları nesnenin bağlamında gerçekleştiğinde çağrılır.

Bazı yaygın HTML olayları burada belirtilmiştir:

Event	Description
`onchange`	Form girişinin değeri değiştirildiğinde veya değiştirildiğinde
`onclick`	Kullanıcı, öğeye tıkladığında
`onmouseover`	Fare imleci öğenin üzerine geldiğinde
`onmouseout`	Fare imleci öğeden ayrıldığında
`onkeydown`	Kullanıcı anahtarı basıp sonra bırakır
`onload`	Tarayıcı yüklemeyi tamamladığında

Çocukları olan düğümlere kaydedilen handlers, çocuklardan da olaylar alabilir. Örneğin, bir paragraf içindeki bir düğme tıklandığında, paragrafta kaydedilen handlers de tıklama olayını alır.

Olay handler, event nesnesi üzerindeki `stopPropagation` yöntemini çağırarak, daha ilerideki handlers'ın olayı almasını engelleyebilir. Bu, örneğin, tıklanabilir bir elemanın içinde bir düğmeniz varsa ve bir düğme tıklamasından dış elemanın tıklanabilir davranışını tetiklemek istemiyorsanız kullanışlıdır.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p"),
      button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Paragraph handler.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Button handler.");
    event.stopPropagation();
  });
</script>
```

Burada, `mousedown` handlers hem paragraf hem de düğme tarafından kaydedilir. Düğmeye tıklandığında, düğme için handler `stopPropagation` çağırır, bu da paragraftaki handler'ın çalışmasını engelleyecektir.

Olayların varsayılan bir davranışı olabilir. Örneğin, bağlantılar, bağlantının hedefine tıklandığında gezinir, aşağı ok düğmesine tıklandığında sayfanın altına yönlendirilir vb. Bu varsayılan davranışlar, event nesnesi üzerindeki `preventDefault` yöntemini çağırarak önlenebilir.

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Burada, tıklandığında bağlantının varsayılan davranışı önlenir, yani bağlantının hedefine yönlendirilmez.

Bölüm 18

Promise, async/await

Bir popüler kitap yazarı olduğunuzu ve belirli bir günde yeni bir kitap yayınlamayı planladığınızı hayal edin. Bu kitaba ilgi duyan okuyucular, bu kitabı listelerine ekliyor ve yayınlandığında veya yayın tarihi ertelendiğinde bile bilgilendiriliyor. Yayın günü geldiğinde, herkes bilgilendirilir ve kitabı satın alabilir ve tüm taraflar mutlu olur. Bu, programlamada gerçekleşen gerçek hayattan bir analojidir.

1. Bir "*producing code*" (*üreten kod*) zaman alan ve bir şeyleri başaran bir şeydir. Burada kitap yazarı.
2. Bir "*consuming code*" (*tüketen kod*) üreten kodu hazır olduğunda tüketen kişidir. Bu durumda "okur"dur.
3. "üreten kod" ile "tüketen kod" arasındaki bağlantı, sonuçları "üreten kod" dan "tüketen kod" a ulaştıracağı için bir sözleşme olarak adlandırılabilir.

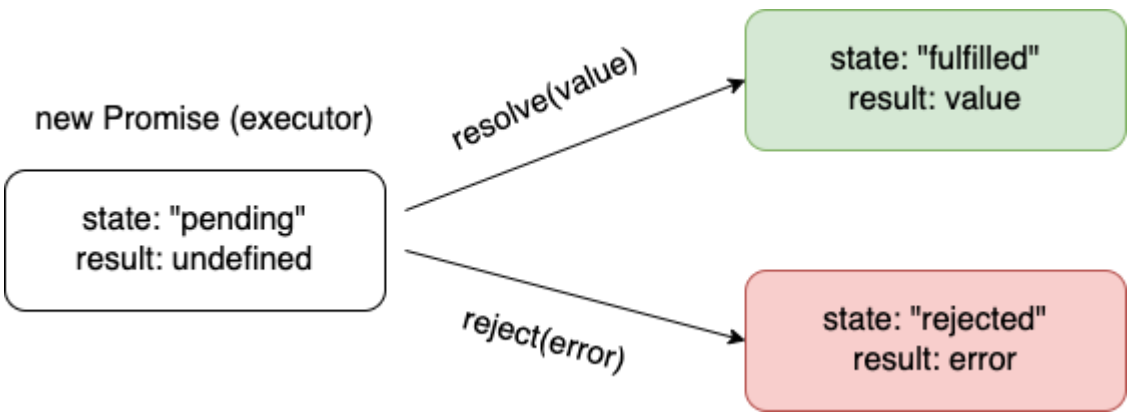
Promise

Yaptığımız analoji, JavaScript promise nesnesi için de geçerlidir. promise nesnesi için kurucu sözdizimi şöyledir:

```
let promise = new Promise(function (resolve, reject) {
  // executor (the producing code, "writer")
});
```

Burada, `new Promise` 'ye bir işlev geçirilir, buna executor da denir ve oluşturulurken otomatik olarak çalışır. Sonuç veren üreten kodu içerir. `resolve` ve `rejects` JavaScript tarafından sağlanan ve sonuçlar üzerine çağrılan argümanlardır.

- `resolve(value):` sonuç üzerine value döndüren bir callback fonksiyonu
- `reject(error)` : hata üzerine `error` döndüren bir callback fonksiyonu, bir error nesnesi döndürür



`new Promise` kurucusu tarafından döndürülen `promise` nesnesinin iç özellikleri şunlardır:

- `state` - başlangıçta `pending` , ardından `fulfill` olmak üzere üzerine resolve veya `rejected` üzerine `reject` çağrıldığında değişir
- result - başlangıçta `undefined` , ardından `resolve` üzerine `value` veya `reject` üzerine `error` değişir

> Promise özelliklerine `state` ve `result` erişilemez. Promise'leri işlemek için promise yöntemlerine ihtiyaç vardır.

Promise örneği:

```
let promiseOne = new Promise(function (resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1-second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});

let promiseTwo = new Promise(function (resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1-second signal that the job is done with the result "error"
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

Burada, `promiseOne` "yerine getirilmiş promise" örneğidir, çünkü değerler başarıyla çözüldüğünden, `promiseTwo` ise reddedildiği için "reddedilmiş promise" örneğidir. Reddedilmiş veya yerine getirilmiş bir promise, yerleşik promise olarak adlandırılır, aksine başlangıçta bekleyen promise. Promise'den tüketen fonksiyon `.then` ve `.catch` yöntemleri kullanılarak kaydedilebilir. Ayrıca, önceki yöntemlerin tamamlanmasından sonra temizleme veya sonlandırma için `.finally` yöntemini de ekleyebilirsiniz.

```
let promiseOne = new Promise(function (resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promiseOne.then(
  (result) => alert(result), // shows "done!" after 1 second
  (error) => alert(error) // doesn't run
);

let promiseTwo = new Promise(function (resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promiseTwo.then(
  (result) => alert(result), // doesn't run
  (error) => alert(error) // shows "Error: Whoops!" after 1 second
);

let promiseThree = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promiseThree.catch(alert); // shows "Error: Whoops!" after 1 second
```

`Promise.then()` metodunda, her iki callback argümanının da kullanımı isteğe bağlıdır.

Async/Await

Async/Await, Promise'leri daha kolay yazmaya yarayan bir JavaScript özelliğidir. `async` anahtar kelimesi, bir Promise döndüren ve `await` sözdizimi, JavaScript'in o Promise'nin çözülmesi ve değerinin döndürülmesini beklemesini sağlar.

```
//async function f
async function f() {
  return 1;
}
// promise being resolved
f().then(alert); // 1
```

Yukarıdaki örnek, aşağıdaki şekilde de yazılabilir:

```
function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

`async` , işlevin bir Promise döndürmesini sağlar ve Promise olmayanları ona sarar. `await` ile, JavaScript'in Promise çözülene ve değeri döndürilene kadar beklemesini sağlayabiliriz.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Welcome to Learn JavaScript!"), 1000);
  });

  let result = await promise; // wait until the promise resolves (*)
  alert(result); // "Welcome to Learn JavaScript!"
}

f();
```

`await` anahtar kelimesi yalnızca `async` fonksiyonları içinde kullanılabilir.

Bölüm 19

Miscellaneous

Bu bölümde, kod yazarken karşılaşabileceğimiz çeşitli konuları tartışacağız. Konular aşağıda listelenmiştir:

- Template Literals
- Hoisting
- Currying
- Polyfills and Transpilers
- Linked List
- Global Footprint
- Debugging

Template Literals

Template literals (``) backtick'i ile ayrıştırılmış literallerdir ve dizelere değişken ve ifade eklemesinde kullanılırlar.

```
let text = `Hello World!`;
// tek bir dize içinde hem tek hem de çift kod içeren template literals
let text = `He's often called "Johnny"`;
// çoklu satırlı template literals
let text = `The quick
brown fox
jumps over
the lazy dog`;

// değişken içeren template literals
const firstName = "John";
const lastName = "Doe";

const welcomeText = `Welcome ${firstName}, ${lastName}!`;

// ifade içeren template literals
const price = 10;
const VAT = 0.25;

const total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
```

Hoisting

Hosting, JavaScript'te tanımlamaları en üste taşımak için varsayılan bir davranıştır. Bir kod yürütülürken, global bir yürütme bağlamı oluşturur: oluşturma ve yürütme. Oluşturma aşamasında, JavaScript değişken ve işlev tanımlamalarını sayfanın en üstüne taşır, bu da hoisting olarak bilinir.

```
// değişken hoisting
console.log(counter);
let counter = 1; // "ReferenceError: Cannot access 'counter' before initialization" hatası verir
```

Her ne kadar `counter` değişkeni bellekte mevcut olsa da değişken ilk değerini almadığı için hata veriyor. Bunun nedeni, `counter` değişkeninin burada sayfanın yukarısına çekilmesidir.

```
// function hoisting
const x = 20,
    y = 10;

let result = add(x,y); // ❌ Uncaught ReferenceError: add is not defined
console.log(result);

let add = (x, y) => x + y;
```

Burada, `add` işlevi, global yürütme bağlamının oluşturulması aşamasında bellekte `undefined` ile başlatılır ve yukarı çekilir. Böylece, bir hata atılır.

Currying

`Currying` fonksiyonel programlamada birden fazla argümanı olan bir fonksiyonu iç içe geçmiş fonksiyonlar dizisine dönüştüren ileri bir tekniktir. Bir fonksiyonu `f(a,b,c)` şeklinde çağrılabilirken `f(a)(b)(c)` şeklinde çağrılabilir hale dönüştürür. Bir fonksiyonu çağırmaz, bunun yerine dönüştürür.

`currying` 'i daha iyi anlamak için üç argüman alan ve bunların toplamını döndüren basit bir `add` fonksiyonu oluşturalım. Ardından, tek bir girdi alan ve toplamıyla birlikte bir dizi fonksiyon döndüren bir `addCurry` fonksiyonu oluşturalım.

```
// Currying olmadan yazılan versiyon
const add = (a, b, c) => {
  return a + b + c;
};
console.log(add(2, 3, 5)); // 10

// currying olan versiyon
const addCurry = (a) => {
  return (b) => {
    return (c) => {
      return a + b + c;
    };
  };
};
console.log(addCurry(2)(3)(5)); // 10
```

Burada, hem curried hem de curried olmayan versiyonların aynı sonucu verdiğini görebiliriz. Currying birçok nedenden dolayı faydalı olabilir, bunlardan bazıları aşağıda belirtilmiştir.

- Aynı değişkeni tekrar tekrar kullanmaktan kaçınmaya yardımcı olur.
- Fonksiyonu tek bir sorumlulukla daha küçük parçalara böler ve fonksiyonu daha az hataya eğilimli hale getirir.
- Fonksiyonel programlamada yüksek dereceli bir fonksiyon (*high order function*) oluşturmak için kullanılır.

Polyfills and Transpilers

JavaScript belirli aralıklarla gelişmektedir. Düzenli olarak yeni dil önerileri sunulur, analiz edilir ve https://tc39.github.io/ecma262/adresine eklenir ve ardından spesifikasyona dahil edilir. Tarayıcıya bağlı olarak JavaScript motorlarında nasıl uygulandığı konusunda farklılıklar olabilir. Bazıları taslak önerileri uygularken, diğerleri tüm spesifikasyon yayınlanana kadar bekleyebilir. Yeni şeyler tanıtıldıkça geriye dönük uyumluluk sorunları ortaya çıkar.

Eski tarayıcılarda modern kodu desteklemek için iki araç kullanırız: `transpilers` ve `polyfills` .

Transpilers

Modern kodu çeviren ve eski sözdizimi yapılarını kullanarak yeniden yazan bir programdır, böylece eski motor bunu anlayabilir. Örneğin, " `nullish` birleştirme operatörü" `??` 2020'de tanıtıldı ve eski tarayıcılar bunu anlayamıyor.

Şimdi, `nullish` birleştirme operatörünü" `??` eski tarayıcılar için anlaşılabilir hale getirmek transpiler'ın işidir.

```
// transpiler'i çalıştırmadan önce
height = height ?? 200;

// transpiler çalıştırıldıktan sonra
height = height !== undefined && height !== null ? height : 200;
```

Babel en önde gelen transpile araçlarından biridir. Geliştirme sürecinde, kodu transpile etmek için webpack veya parcel gibi derleme araçlarını kullanabiliriz.

Polyfills

Yeni fonksiyonların eski tarayıcı motorlarında mevcut olmadığı durumlar vardır. Bu durumda, yeni fonksiyonu kullanan kod çalışmayacaktır. Boşlukları doldurmak için, `polyfill` olarak adlandırılan eksik fonksiyonu ekleriz. Örneğin, `filter()` yöntemi ES5'te tanıtıldı ve bazı eski tarayıcılarda desteklenmiyor. Bu yöntem bir fonksiyon kabul eder ve sadece fonksiyonun `true` döndürdüğü orijinal dizinin değerlerini içeren bir dizi döndürür.

```
const arr = [1, 2, 3, 4, 5, 6];
const filtered = arr.filter((e) => e % 2 === 0); // çift sayıyı filtreler
console.log(filtered);

// [2, 4, 6]
```

`filter` fonksiyonu için polyfill şu şekildedir.

```
Array.prototype.filter = function (callback) {
  //  Yeni diziyi saklar
  const result = [];
  for (let i = 0; i < this.length; i++) {
    // callback'i geçerli öğe, dizin ve bağlam ile çağırır.
    // verilen şartı sağlarsa yeni diziye ekler.
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }
  // diziyi döndürür
  return result;
};
```

caniuse farklı tarayıcı motorları tarafından desteklenen güncellenmiş fonksiyonları ve sözdizimini gösterir.

Linked List

Tüm programlama dillerinde bulunan yaygın bir veri yapısıdır. Linked List, Javascript'teki normal bir diziye çok benzer, sadece biraz farklı davranır.

Burada listedeki her bir öğe, bir sonraki öğeye bir bağlantı veya işaretçi içeren ayrı bir nesnedir. Javascript'te Linked List'ler için yerleşik bir method ya da fonksiyon yoktur, bu yüzden bunu kendiniz uygulamanız gerekir. Aşağıda bir Linked List örneği gösterilmektedir.

```
["one", "two", "three", "four"]
```

Linked List Türleri

Üç farklı türde Linked List vardır:

1. **Tek Linked List:** Her node bir sonraki node için sadece bir işaretçi içerir.
2. **Çift Linked List:** Her node'da, biri bir sonraki node'u diğeri de bir önceki node'u giden iki işaretçi vardır.
3. **Dairesel Linked Lists:** Dairesel Linked List, son node'un ilk node'a veya ondan önceki herhangi bir node'a işaret etmesiyle bir döngü oluşturur.

Add

Burada, linked list'e değer eklemek için `add` yöntemi kullanılır.

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor(head) {
    this.head = head;
  }
  append = (value) => {
    const newNode = new Node(value);
    let current = this.head;
    if (!this.head) {
      this.head = newNode;
      return;
    }
    while (current.next) {
      current = current.next;
    }
    current.next = newNode;
  };
}
```

Pop

Burada, linked list'den bir değeri kaldırmak için bir `pop` yöntemi kullanılır.

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor(head) {
    this.head = head;
  }
  pop = () => {
    let current = this.head;
    while (current.next.next) {
      current = current.next;
    }
    current.next = current.next.next;
  };
}
```

Prepend

Burada, linked list'in ilk elemanından önce bir değer eklemek için bir `prepend` yöntemi kullanılır.

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor(head) {
    this.head = head;
  }
  prepend = (value) => {
    const newNode = new Node(value);
    if (!this.head) {
      this.head = newNode;
    } else {
      newNode.next = this.head;
      this.head = newNode;
    }
  };
}
```

Shift

Burada, Linked List'in ilk elemanını kaldırmak için `shift` yöntemi kullanılır.

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor(head) {
    this.head = head;
  }
  shift = () => {
    this.head = this.head.next;
  };
}
```

Global footprint

Başka modüllerin de çalıştığı bir web sayfasında çalışabilecek bir modül geliştiriyorsanız, değişken adlarının çakışmasına dikkat etmeniz gerekir.

Bir sayaç modülü geliştirdiğimizi varsayalım:

```
let myCounter = {
  number: 0,
  plusPlus: function () {
    this.number = this.number + 1;
  },
  isGreaterThanTen: function () {
    return this.number > 10;
  },
};
```

> ***Note:*** Bu teknik, dahili durumu dışarıdan değişmez kılmak için genellikle closure'larla birlikte kullanılır.

Modül artık sadece bir değişken adı alıyor - `myCounter`. Eğer sayfadaki başka bir modül `number` veya `isGreaterThanTen` gibi isimler kullanıyorsa, bu tamamen güvenlidir çünkü birbirlerinin değerlerini geçersiz kılmayacaklardır.

Debugging (Hata Ayıklama)

Programlamada, kod yazarken hatalar oluşabilir. Bu, sözdizimsel veya mantıksal hatalardan kaynaklanabilir. Hata bulma süreci zaman alıcı ve zor olabilir ve kod hata ayıklama olarak adlandırılır.

Neyse ki, modern tarayıcıların çoğunda yerleşik hata ayıklayıcılar bulunmaktadır. Bu hata ayıklayıcılar açılıp kapatılabilir ve hataların bildirilmesi sağlanabilir. Kodun yürütülmesi sırasında yürütmeyi durdurmak ve değişkenleri incelemek için kesme noktaları ayarlamak da mümkündür. Bunun için bir hata ayıklama penceresi açmak ve JavaScript koduna `debugger` anahtar sözcüğünü yerleştirmek gerekir. Kodun yürütülmesi her breakpoint'te durdurulur ve geliştiricilerin JavaScript değerlerini incelemesine ve kodun yürütülmesine devam etmesine olanak tanır.

JavaScript değerlerini hata ayıklayıcı penceresine yazdırmak için `console.log()` yöntemi de kullanılabilir.

```
const a = 5,
  b = 6;
const c = a + b;
console.log(c);
// Result : c = 11;
```

Bölüm 20

Exercises (Alıştırmalar)

Bu bölümde JavaScript bilgimizi test etmek için alıştırmalar yapacağız. Yapacağımız alıştırmalar aşağıda listelenmiştir:

- Console (Konsol)
- Multiplication(Çarpma İşlemi)
- User Input Variables (Kullanıcı Girdisi Değişkenleri)
- Constants (Sabitler)
- Concatenation (Birleştirme)
- Functions (Fonksiyonlar)
- Conditional Statements (Koşullu İfadeler)
- Objects (Objeler)
- FizzBuzz Problem
- Get the Titles!

Console (Konsol)

JavaScript'te, `console`'a bir mesaj(bir değişkenin içeriği, verilen bir dize, vb.) yazmak için `console.log()` işlevini kullanırız. Genellikle hata ayıklama amacıyla, program çalışırken değişkenlerin içeriğinin bir kopyasını göstermek için kullanılır.

Örnek:

```
console.log("Welcome to Learn JavaScript Beginners Edition");
let age = 30;
console.log(age);
```

📝 Alıştırmalar:

- [ ] Konsola `Hello World` yazdıran bir program yazın. Başka şeyleri de yazdırmaktan çekinmeyin!
- [ ] `console`'a değişkenleri yazdıran bir program yazın.
    1. `animal` adında bir değişken tanımlayın ve değer olarak "dragon" atayın.
    2. `animal` değişkenini konsol'a yazdırın..

💡 İpucu:

- Değişkenler bölümüne göz atın.

Multiplication (Çarpma İşlemi)

Javascript'te yıldız işareti(`*`) kullanarak iki sayıyı çarpabiliriz.

Örnek:

```
let resultingValue = 3 * 2;
```

Burada, `3 * 2` çarpımını `resultingValue` adında bir değişkende sakladık.

📝 Alıştırma:

- [ ] `23` ve `41` çarpımının değerini yazdıran bir program yazın.

💡 İpucu:

- Matematiksel işlemleri anlamak için Temel Operatörler bölümünü ziyaret edin.

User Input Variables (Kullanıcı Girdisi Değişkenleri)

JavaScript'te kullanıcılardan girdi alabilir ve bunu bir değişken olarak kullanabiliriz. Onlarla işlem yapmak için değerlerini bilmek gerekmez.

📝 Alıştırma:

- [ ] Kullanıcıdan bir girdi alan ve buna `10` ekleyip sonucu gösteren bir program yazın.

💡 İpuçları:

- Bir değişkenin içeriği kullanıcının girdileri tarafından belirlenir. `prompt()` fonksiyonu girdi değerini bir string (*dize*) olarak kaydeder.
- string(*dize*) değerini öncelikle bir sayıya dönüştürmeniz gerekecektir.
- String `in` int`e tür dönüşümü için Temel Operatörler bölümünü ziyaret edin.

Constants (Sabitler)

Constants ES6(2015)'te tanıtılmıştır ve `const` anahtar sözcüğünü kullanır. `const` ile tanımlanan değişkenler yeniden atanamaz veya yeniden tanımlanamaz.

Örnek:

```
const VERSION = "1.2";
```

📝 Alıştırma:

- [ ] Aşağıda belirtilen programı çalıştırın ve konsolda gördüğünüz hatayı düzeltin. Konsolda düzeltildiğinde kod sonucunun `0.9` olduğundan emin olun.

```
const VERSION = "0.7";
VERSION = "0.9";
console.log(VERSION);
```

💡 İpucu:

`const` hakkında daha fazla bilgi için Değişkenler bölümünü ziyaret edin ve ayrıca bir bu hatayaı öğrenmek için arama motorlarında "*TypeError assignment to constant variable*" ifadesini arayın.

Concatenation (Birleştirme)

Herhangi bir programlama dilinde, dize(*string*) birleştirme basitçe bir veya daha fazla dizenin başka bir dizeye eklenmesi anlamına gelir. Örneğin, "*World*" ve "*Good Afternoon*" dizeleri "*Hello*" dizesiyle birleştirildiğinde, "*Hello World, Good Afternoon*" dizesini oluştururlar. JavaScript'te bir dizeyi çeşitli şekillerde birleştirebiliriz.

Örnek:

```
const icon = "👋";

// template literals  kullarak
`hi ${icon}`;

// join() fonksiyonunu kullanarak
["hi", icon].join(" ");

// concat() fonksiyonunu kullanarak
"".concat("hi ", icon);

//  + operatörünü kulllanarak
"hi " + icon;

// Sonuç
// hi 👋
```

📝 Alıştırma:

- [ ] Konsola '*Hello World*' yazdırması için `str1` ve `str2` değerlerini kullanan bir program yazın.

💡 İpucu:

- Dize birleştirme hakkında daha fazla bilgi için dizelerin birleştirme bölümünü ziyaret edin.

Functions (Fonksiyonlar)

Fonksiyon, belirli bir görevi yerine getirmek üzere tasarlanmış ve "bir şey" onu çağırdığında çalıştırılan bir kod bloğudur. Fonksiyonlar hakkında daha fazla bilgi fonksiyonlar bölümünde bulunabilir.

📝 Alıştırma:

- [ ] Bir argüman olarak `45345` sayısını alan ve sayının tek olup olmadığını belirleyen `isOdd` adlı bir fonksiyon oluşturun.
- [ ] Sonucu almak için bu fonksiyonu çağırın. Sonuç boolean formatında olmalı ve `console`'da `true` döndürmelidir.

💡 İpucu:

- Fonksiyonlar hakkında daha fazla bilgi için fonksiyonlar bölümünde göz atın.

Conditional Statements (Koşul İfadeleri)

Koşullu mantık programlamada hayati önem taşır çünkü programa attığınız verilerden bağımsız olarak programın çalışmasını sağlar ve ayrıca dallanmaya izin verir. Bu alıştırma, gerçek hayat problemlerinde çeşitli koşullu mantığın uygulanmasıyla ilgilidir.

📝 Alıştırma:

- [ ] "*Kaç km kaldı?"* sorusunu oluşturacak ve kullanıcıya ve aşağıdaki koşullara bağlı olarak sonuçları `console` 'a yazdıracak bir program yazın.
  - Eğer 100 km'den fazla yol kaldıysa: `Hala gidecek biraz yolunuz var` yazdırın.
  - Eğer 50 km'den fazla ama 100 km'den az veya eşitse `5 dakika içinde orada olacağım` yazdırın.
  - Eğer 50 km'den az veya eşitse `Park ediyorum. Şimdi görüşürüz` yazdırın.

💡 İpucu:

- Koşullu mantığın ve koşullu ifadelerin nasıl kullanılacağını anlamak için koşullu mantık bölümünü ziyaret edin.

Objects (Objeler)

Objeler `anahtar` , `değer` çiftlerinin koleksiyonudur ve her anahtar-değer (*key-value*) çifti bir özellik olarak bilinir.

📝 Alıştırmalar:

İki üyeli bir Doe ailesi verildiğinde, her üyenin bilgisi bir obje şeklinde sunulur.

```
let person = {
  name: "John", //String
  lastName: "Doe",
  age: 35, //Number
  gender: "male",
  luckyNumbers: [7, 11, 13, 17], //Array
  significantOther: person2, //Object,
};

let person2 = {
  name: "Jane",
  lastName: "Doe",
  age: 38,
  gender: "female",
  luckyNumbers: [2, 4, 6, 8],
  significantOther: person,
};

let family = {
  lastName: "Doe",
  members: [person, person2], //Array of objects
};
```

- [ ] Doe ailesinin ilk üyesinin adını `console` 'da yazdırmanın bir yolunu bulun.
- [ ] Doe ailesinin ikinci üyesinin `luckyNumbers` 'in dördüncü değerini `33` olarak değiştirin.
- [ ] Doe ailesine yeni bir kişi ekleyin(`Jimmy`, `Doe` , `13` , `male` , `[1, 2, 3, 4]` , `null`) ve `family` objesini güncelleyin.
- [ ] Doe ailesinin şanslı sayılarının toplamını `console` 'a yazdırın.

💡 İpuçları:

- Objelerin nasıl çalıştığını anlamak için objeler bölümünü ziyaret edin.
- Aile objesi içindeki her bir kişi objesinden `luckyNumbers` alabilirsiniz.
- Diziyi aldıktan sonra,her elemanı ekleyen bir döngü oluşturun ve ardından 3 aile üyesinin her bir toplamını ekleyin.

FizzBuzz Problem

FizzBuzz problemi sıkça sorulan sorulardan biridir, bu problemde bazı koşullar altında *Fizz* ve *Buzz* yazdırmamız istenir.

📝 Alıştırma:

- [ ] 1'den 100'e kadar olan tüm sayıları aşağıdaki koşulları sağlayacak şekilde yazdıracak bir program yazınız.
  - 3'ün katları için, sayı yerine `Fizz` yazdırın.
  - 5'in katları için `Buzz` yazdırın.
  - Hem 3 hem de 5'in katı olan sayılar için `FizzBuzz` yazdırın.

```
/
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
....
....
98
Fizz
Buzz
/
```

💡 İpucu:

- Döngüleri daha iyi anlamak için döngüler bölümüne bakın.

Get the Titles! (Başlıkları al!)

Başlıkları Al! problemi, bir kitap listesinden başlığı almamız gereken eğlenceli bir problemdir. Bu problem, dizilerin ve objelerin kullanımı için güzel bir alıştırmadır.

📝 Alıştırma:

Kitaplar ve yazarlarından oluşan bir array (*dize*) verilmiştir.

```
const books = [
  {
    title: "Eloquent JavaScript, Third Edition",
    author: "Marijn Haverbeke",
  },
  {
    title: "Practical Modern JavaScript",
    author: "Nicolás Bevacqua",
  },
];
```

- [ ] Parametre olarak dizi alan ve başlıkları bir dizi içerisinde döndüren `getTheTitles` adında bir fonksiyon oluşturun.

💡 İpucu:

- Dizi ve objelerin nasıl çalıştığını anlamak için diziler ve objeler bölümlerini ziyaret edin.

References

Ballard, P. (2018). JavaScript in 24 Hours, Sams Teach Yourself. Sams Publishing.

Crockford, D. (2008). JavaScript: The Good Parts. O'Reilly Media.

Duckett, J. (2011). HTML & CSS: Design and Build Websites. Wiley.

Duckett, J. (2014). JavaScript and JQuery: Interactive Front-End Web Development. Wiley.

Flanagan, D. (2011). JavaScript: The Definitive Guide. O'Reilly Media.

Freeman, E., & Robson, E. (2014). Head First JavaScript Programming: A Brain-Friendly Guide. O'Reilly Media.

Frisbie, M. (2019). Professional JavaScript for Web Developers. Wrox.

Haverbeke, M. (2019). Eloquent JavaScript: A Modern Introduction to Programming. No Starch Press.

Herman, D. (2012). Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript. Addison-Wesley Professional.

McPeak, J., & Wilton, P. (2015). Beginning JavaScript. Wiley.

Morgan, N. (2014). JavaScript for Kids: A Playful Introduction to Programming. No Starch Press.

Murphy C, Clark R, Studholme O, et al. (2014). Beginning HTML5 and CSS3: The Web Evolved. Apress.

Nixon, R. (2021). Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5. O'Reilly Media.

Powell, T., & Schneider, F. (2012). JavaScript: The Complete Reference. McGraw-Hill Education.

Resig, J. (2007). Pro JavaScript Techniques. Apress.

Resig, J., & Bibeault, B. (2016). Secrets of the JavaScript Ninja. Manning Publications.

Robbins, J. N. (2014). Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics. O'Reilly Media.