İlgili konular

İfadeler ve işleçler

Bu bölümde JavaScript'in atama, karşılaştırma, aritmetik, bitsel, mantıksal, dizi, üçlü ve daha fazlası dahil olmak üzere ifadeleri ve işleçleri açıklanmaktadır.

Yüksek düzeyde *ifade* , bir değere çözümlenen geçerli bir kod birimidir. İki tür ifade vardır: yan etkileri olan (değer atama gibi) ve tamamen değerlendiren *ifadeler* .

İfade, `x = 7` birinci türün bir örneğidir. Bu ifade, yedi değerini değişkene atamak için `=` *operatörü* `x` kullanır . İfadenin kendisi olarak değerlendirilir `7` .

İfade, `3 + 4` ikinci türün bir örneğidir. Bu ifade, ve birlikte `+` eklemek için işleci kullanır ve bir değer üretir, . Ancak, sonunda daha büyük bir yapının parçası değilse (örneğin, gibi bir değişken bildirimi), sonucu hemen atılacaktır - bu genellikle bir programcı hatasıdır çünkü değerlendirme herhangi bir etki üretmez. `3 4 7 const z = 3 + 4`

Yukarıdaki örneklerin de gösterdiği gibi, tüm karmaşık ifadeler ve gibi *işleçlerle* birleştirilir . Bu bölümde, aşağıdaki operatörleri tanıtacağız: `= +`

- Atama işleçleri

- Karşılaştırma işleçleri
- Aritmetik operatörler
- bitsel operatörler
- Mantıksal operatörler
- BigInt operatörleri
- Dize işleçleri
- Koşullu (üçlü) işleç
- virgül operatörü
- tekli operatörler
- ilişki operatörleri

Bu işleçler, ya daha yüksek öncelikli işleçler ya da temel ifadelerden biri tarafından oluşturulan işlenenleri birleştirir . İşleçlerin ve ifadelerin eksiksiz ve ayrıntılı bir listesi de referansta mevcuttur .

İşleçlerin önceliği , bir ifade değerlendirilirken uygulandıkları sırayı belirler. Örneğin:

```
const x = 1 + 2 * 3;
const y = 2 * 3 + 1;
```

`*` ve farklı sıralarda gelmesine rağmen `+` , her iki ifade de şuna neden olur `7` çünkü `*` önceliği vardır `+` , bu nedenle `*` -joined ifadesi her zaman önce değerlendirilir. Parantez kullanarak operatör önceliğini geçersiz kılabilirsiniz (bu, gruplandırılmış bir ifade oluşturur - temel ifade). Çeşitli uyarıların yanı sıra eksiksiz bir operatör önceliği tablosunu görmek için Operatör Önceliği Referansı sayfasına bakın.

JavaScript'in hem *ikili* hem de *tekli* işleçleri ve özel bir üçlü işleci, koşullu işleci vardır. Bir ikili operatör, biri

operatörden önce ve diğeri operatörden sonra olmak üzere iki işlenen gerektirir:

```
operand1 operator operand2
```

Örneğin `3 + 4` veya `x * y`. Operatör iki işlenen arasına yerleştirildiğinden, bu forma *ekli* ikili operatör denir. JavaScript'teki tüm ikili işleçler infix'tir.

Tekli bir operatör, operatörden önce veya sonra tek bir işlenen gerektirir:

```
operator operand
operand operator
```

Örneğin `x++` veya `++x`. Forma *önek* `operator operand` tekli işleç denir ve forma *sonek tekli* işleç denir. ve JavaScript'teki tek son ek işleçleridir — , , vb. gibi diğer tüm işleçler önektir. `operand operator ++ -- ! typeof`

Atama işleçleri

Bir atama işleci, sağ işleneninin değerine bağlı olarak sol işlenenine bir değer atar. `=` Sağ işleneninin değerini sol işlenenine atayan basit atama operatörü eşittir ( ). Yani, to `x = f()` değerini atayan bir atama ifadesidir. `f() x`

Aşağıdaki tabloda listelenen işlemler için kısayol olan bileşik atama işleçleri de vardır:

İsim	steno operatörü	Anlam
Atama	`x = f()`	`x = f()`

İsim	steno operatörü	Anlam
ek atama	`x += f()`	`x = x + f()`
çıkarma atama	`x -= f()`	`x = x - f()`
Çarpma ataması	`x *= f()`	`x = x * f()`
Bölüm ataması	`x /= f()`	`x = x / f()`
Kalan atama	`x %= f()`	`x = x % f()`
Üs ataması	`x **= f()`	`x = x ** f()`
Sola kaydırma ataması	`x <<= f()`	`x = x << f()`
Sağ vardiya ataması	`x >>= f()`	`x = x >> f()`
İmzasız sağa kaydırma ataması	`x >>>= f()`	`x = x >>> f()`
bit düzeyinde AND ataması	`x &= f()`	`x = x & f()`
Bit düzeyinde XOR ataması	`x ^= f()`	`x = x ^ f()`
Bit düzeyinde VEYA ataması	`x |= f()`	`x = x | f()`
Mantıksal AND ataması	`x &&= f()`	`x && (x = f())`

İsim	steno operatörü	Anlam
Mantıksal VEYA ataması	`x ||= f()`	`x || (x = f())`
Nullish birleştirme ataması	`x ??= f()`	`x ?? (x = f())`

Özelliklere atama

Bir ifade bir nesne olarak değerlendirilirse , atama ifadesinin sol tarafı o ifadenin özelliklerine atamalar yapabilir. Örneğin:

```
const obj = {};

obj.x = 3;
console.log(obj.x); // Prints 3.
console.log(obj); // Prints { x: 3 }.

const key = "y";
obj[key] = 5;
console.log(obj[key]); // Prints 5.
console.log(obj); // Prints { x: 3, y: 5 }.
```

Nesneler hakkında daha fazla bilgi için Nesnelerle Çalışma bölümünü okuyun .

Bir ifade bir nesne olarak değerlendirilmezse, o ifadenin özelliklerine yapılan atamalar şunları atamaz:

```
const val = 0;
val.x = 3;

console.log(val.x); // Prints undefined.
console.log(val); // Prints 0.
```

Katı modda , yukarıdaki kod atar, çünkü ilkellere özellikler atanamaz.

`null` Değiştirilemez özelliklere veya özellikler (veya) olmadan bir ifadenin özelliklerine değer atamak bir hatadır `undefined` .

Şeklini bozma

Daha karmaşık atamalar için, yapıyı bozan atama sözdizimi, dizi ve nesne sabit değerlerinin yapısını yansıtan bir sözdizimi kullanarak dizilerden veya nesnelerden veri çıkarmayı mümkün kılan bir JavaScript ifadesidir.

```
const foo = ['one', 'two', 'three'];

// without destructuring
const one   = foo[0];
const two   = foo[1];
const three = foo[2];

// with destructuring
const [one, two, three] = foo;
```

Değerlendirme ve yerleştirme

Genel olarak, atamalar bir değişken bildirimi içinde (yani `const` , `let` , veya `var`) veya bağımsız ifadeler olarak kullanılır.

```
// Declares a variable x and initializes it to the
result of f().
// The result of the x = f() assignment expression is
discarded.
let x = f();

x = g(); // Reassigns the variable x to the result of
g().
```

Ancak, diğer ifadeler gibi atama ifadeleri de `x = f()` bir sonuç değeri olarak değerlendirilir. Bu sonuç değeri genellikle kullanılmasa da, daha sonra başka bir ifade tarafından kullanılabilir.

Diğer ifadelerde zincirleme atamalar veya iç içe atamalar şaşırtıcı davranışlara neden olabilir. Bu nedenle, bazı JavaScript stil kılavuzları [zincirleme veya iç içe atamaları önermez]( ). Bununla birlikte, atama zinciri ve yerleştirme bazen meydana gelebilir, bu nedenle bunların nasıl çalıştığını anlamak önemlidir.

Bir atama ifadesini zincirleyerek veya iç içe geçirerek, sonucunun kendisi başka bir değişkene atanabilir. Günlüğe kaydedilebilir, bir dizi sabit değeri veya işlev çağrısı içine yerleştirilebilir, vb.

```
let x;
const y = (x = f()); // Or equivalently: const y = x =
f();
console.log(y); // Logs the return value of the
assignment x = f().

console.log(x = f()); // Logs the return value
directly.

// An assignment expression can be nested in any place
// where expressions are generally allowed,
// such as array literals' elements or as function
calls' arguments.
console.log([ 0, x = f(), 0 ]);
console.log(f(0, x = f(), 0));
```

`=` Değerlendirme sonucu , yukarıdaki tablonun "Anlam" sütunundaki işaretin sağındaki ifadeyle eşleşir . Bu , sonucu `x = f()` ne olursa olsun değerlendirir , sonuç toplamı olarak değerlendirir , sonuç gücü olarak değerlendirir , vb. anlamına gelir . `f() x += f() x + f() x **= f() x ** y`

Mantıksal atamalar durumunda, `x &&= f()`, `x ||= f()` ve `x ??= f()`, dönüş değeri atama olmadan mantıksal işlemin değeridir, yani sırasıyla `x && f()`, `x || f()` ve `x ?? f()`.

Bu ifadeleri parantezler veya dizi sabit değerleri gibi diğer gruplama işleçleri olmadan zincirlerken, atama ifadeleri **sağdan sola gruplanır** (bunlar sağla ilişkilendirilebilir), ancak **soldan sağa değerlendirilirler** .

Kendisi dışındaki tüm atama işleçleri için `=` sonuç değerlerinin her zaman işlenenlerin işlemden *önceki değerlerini temel aldığını unutmayın.*

Örneğin, aşağıdaki işlevlerin `f` ve `g` ve değişkenlerinin `x` ve `y` bildirildiğini varsayalım:

```
function f () {
  console.log('F!');
  return 2;
}
function g () {
  console.log('G!');
  return 3;
}
let x, y;
```

Şu üç örneği ele alalım:

```
y = x = f()
y = [ f(), x = g() ]
x[f()] = g()
```

Değerlendirme örneği 1

`y = x = f()` `y = (x = f())` atama işleci sağ-ilişkisel `=` olduğu için , ile eşdeğerdir . Ancak, soldan sağa doğru

değerlendirir:

1. Atama ifadesi `y = x = f()` değerlendirilmeye başlar.
    i. Bu `y` atamanın sol tarafındaki, adlı değişkene bir referans olarak değerlendirilir `y`.
    ii. Atama ifadesi `x = f()` değerlendirilmeye başlar.
        i. Bu `x` atamanın sol tarafındaki, adlı değişkene bir referans olarak değerlendirilir `x`.
        ii. İşlev çağrısı `f()` "F!" konsola ve ardından sayı olarak değerlendirilir `2`.
        iii. Bu `2` sonuç `f()` şuna atanır `x`:.
    iii. Atama ifadesinin `x = f()` değerlendirilmesi artık tamamlandı; `x` bunun sonucu , olan yeni değerdir `2`.
    iv. Bu `2` sonuç sırayla ayrıca atanır `y`.
2. Atama ifadesinin `y = x = f()` değerlendirilmesi artık tamamlandı; sonucu yeni değerdir `y` – ki bu da olur `2`. `x` ve `y` atanmıştır `2` ve konsol "F!" yazdırmıştır.

Değerlendirme örneği 2

`y = [ f(), x = g() ]` ayrıca soldan sağa doğru değerlendirir:

1. Atama ifadesi `y = [ f(), x = g() ]` değerlendirilmeye başlar.
    i. Bu atamanın solundaki `y`, adlı değişkene bir referans olarak değerlendirilir `y`.
    ii. İç dizi hazır değeri `[ f(), x = g() ]` değerlendirmeye başlar.

i. İşlev çağrısı `f()` "F!" konsola ve ardından sayı olarak değerlendirilir `2` .

ii. Atama ifadesi `x = g()` değerlendirilmeye başlar.

i. Bu `x` atamanın sol tarafındaki, adlı değişkene bir referans olarak değerlendirilir `x` .

ii. İşlev çağrısı `g()` "G!" konsola ve ardından sayı olarak değerlendirilir `3` .

iii. Bu `3` sonuç `g()` şuna atanır `x` : .

iii. Atama ifadesinin `x = g()` değerlendirilmesi artık tamamlandı; `x` bunun sonucu , olan yeni değerdir `3` . Bu `3` sonuç, iç dizi hazır bilgisindeki (`2` from ' dan sonra `f()`) bir sonraki öğe olur.

iii. İç dizi hazır bilgisi `[ f(), x = g() ]` artık değerlendirmeyi bitirmiştir; sonucu iki değerli bir dizidir: `[ 2, 3 ]` .

iv. Bu `[ 2, 3 ]` dizi artık `y` .

2. Atama ifadesinin `y = [ f(), x = g() ]` değerlendirilmesi artık tamamlandı; sonucu yeni değerdir `y` – ki bu da olur `[ 2, 3 ]` . `x` şimdi atanmıştır `3` , `y` şimdi atanmıştır `[ 2, 3 ]` ve konsol "F!" yazdırmıştır. sonra "G!"

Değerlendirme örneği 3

`x[f()] = g()` soldan sağa da değerlendirir. (Bu örnek, bunun `x` zaten bir nesneye atanmış olduğunu varsayar. Nesneler hakkında daha fazla bilgi için, bkz. Working with Objects .)

1. Atama ifadesi `x[f()] = g()` değerlendirilmeye başlar.
    i. `x[f()]` Bu atamanın sol tarafındaki mülk erişimi değerlendirilmeye başlar .
        i. Bu özellikteki erişim `x` , adlı değişkene bir başvuru olarak değerlendirilir `x` .
        ii. Ardından işlev çağrısı `f()` "F!" konsola ve ardından sayı olarak değerlendirilir `2` .
    ii. Bu atamadaki mülk `x[f()]` erişiminin değerlendirilmesi artık tamamlandı; sonucu bir değişken özellik başvurusudur: `x[2]` .
    iii. Ardından işlev çağrısı `g()` "G!" konsola ve ardından sayı olarak değerlendirilir `3` .
    iv. Bu `3` şimdi atanan `x[2]` . (Bu adım, yalnızca bir nesneye `x` atanırsa başarılı olur .)
2. Atama ifadesinin `x[f()] = g()` değerlendirilmesi artık tamamlandı; sonucu yeni değerdir `x[2]` – ki bu da olur `3` . `x[2]` şimdi atanmıştır `3` ve konsol "F!" yazdırmıştır. sonra "G!"

Atama zincirlerinden kaçının

Diğer ifadelerde zincirleme atamalar veya iç içe atamalar şaşırtıcı davranışlara neden olabilir. Bu nedenle, aynı ifadede zincirleme atamalar önerilmez).

Özellikle, bir , veya deyimine bir değişken zinciri `const` koymak `let` genellikle `var` işe *yaramaz* . Yalnızca en dıştaki/en soldaki değişken bildirilir; atama zincirindeki diğer değişkenler / / deyimi ile *bildirilmez* . Örneğin: `const let var`

```
const z = y = x = f();
```

`x` Bu ifade görünüşte , `y` ve değişkenlerini bildirir `z` . Ancak, yalnızca gerçekte değişkeni bildirir `z` . `y` ve `x` ya var olmayan değişkenlere geçersiz referanslardır (katı modda) ya da daha kötüsü özensiz modda ve için dolaylı olarak genel değişkenler oluşturur . `x` `y`

Karşılaştırma işleçleri

Bir karşılaştırma operatörü, işlenenlerini karşılaştırır ve karşılaştırmanın doğru olup olmadığına bağlı olarak mantıksal bir değer döndürür. İşlenenler sayısal, dize, mantıksal veya nesne değerleri olabilir. Dizeler, Unicode değerleri kullanılarak standart sözlük sıralamasına göre karşılaştırılır. Çoğu durumda, iki işlenen aynı türden değilse, JavaScript bunları karşılaştırma için uygun bir türe dönüştürmeye çalışır. Bu davranış genellikle işlenenlerin sayısal olarak karşılaştırılmasına neden olur. Karşılaştırmalarda tür dönüştürmenin tek istisnası, `===` ve `!==` katı eşitlik ve eşitsizlik karşılaştırmaları yapan operatörler. Bu operatörler, eşitliği kontrol etmeden önce işlenenleri uyumlu tiplere dönüştürmeye çalışmazlar. Aşağıdaki tablo, karşılaştırma işleçlerini bu örnek kod açısından açıklamaktadır:

```
const var1 = 3;
const var2 = 4;
```

Karşılaştırma işleçleri

Şebeke	Tanım	true dönen örnekler
Eşit (`==`)	`true` İşlenenler eşitse döndürür .	`3 == var1` `"3" == var1`

Şebeke	Tanım	true dönen örnekler
		`3 == '3'`
Eşit değil (`!=`)	`true` İşlenenler eşit değilse döndürür .	`var1 != 4` `var2 != "3"`
Kesin eşittir (`===`)	`true` İşlenenlerin eşit ve aynı türde olup olmadığını döndürür . Ayrıca bkz . JS'de `Object.is` aynılık .	`3 === var1`
Kesin eşit değil (`!==`)	`true` İşlenenler aynı türdeyse ancak eşit değilse veya farklı türdeyse döndürür .	`var1 !== "3"` `3 !== '3'`
(`>`) ' dan büyük	`true` Sol işlenen sağ işlenenden büyükse döndürür .	`var2 > var1` `"12" > 2`
Büyük veya eşittir (`>=`)	`true` Sol işlenen sağ işlenenden büyük veya ona eşitse döndürür .	`var2 >= var1` `var1 >= 3`
(`<`) 'dan az	`true` Sol işlenen sağ işlenenden küçükse döner .	`var1 < var2` `"2" < 12`
Küçük veya	`true` Sol işlenen sağ işlenenden küçük veya ona eşitse döndürür .	`var1 <= var2` `var2 <= 5`

Şebeke	Tanım	true dönen örnekler
eşittir (<=)		

> **Not: bir karşılaştırma işleci değil,** Ok fonksiyonlarının `=>` notasyonudur .

Aritmetik operatörler

Bir aritmetik işleç, işlenenleri olarak sayısal değerleri (sabit değerler veya değişkenler) alır ve tek bir sayısal değer döndürür. Standart aritmetik operatörler toplama (`+`), çıkarma (`-`), çarpma (`*`) ve bölmedir (`/`). Bu işleçler, kayan noktalı sayılarla kullanıldığında diğer programlama dillerinin çoğunda olduğu gibi çalışır (özellikle, sıfıra bölmenin `Infinity`) ürettiğine dikkat edin. Örneğin:

```
1 / 2; // 0.5
1 / 2 === 1.0 / 2.0; // this is true
```

`+` Standart aritmetik işlemlere (, `-` , `*` ,) ek olarak `/` JavaScript, aşağıdaki tabloda listelenen aritmetik işleçleri sağlar:

Aritmetik operatörler

Şebeke	Tanım	Örnek
Kalan (`%`)	İkili operatör. İki işleneni bölmenin tamsayı kalanını döndürür.	%12 5 2 verir.

Şebeke	Tanım	Örnek
Arttırma (`++`)	Tek operatör. İşlenenine bir ekler. Ön ek işleci ( ) olarak kullanılırsa `++x` , bir işlenen ekledikten sonra işleneninin değerini döndürür; sonek işleci ( ) olarak kullanılırsa `x++` , bir işlenen eklemeden önce işleneninin değerini döndürür.	`x` 3 ise, o zaman 4'e ayarlar ve 4'e dönerken, `++x` 3'e döner ve ancak o zaman 4'e ayarlar. `x x++ x`
Azalt (`--`)	Tek operatör. İşleneninden bir çıkarır. Dönüş değeri, artırma işlecininkine benzer.	`x` 3 ise, o zaman 2'ye ayarlar ve 2'ye dönerken, `--x` 3'e döner ve ancak o zaman 2'ye ayarlar. `x x-- x`
Tekli olumsuzlama (`-`)	Tek operatör. İşleneninin olumsuzlamasını döndürür.	`x` 3 ise, -3 - `x` döndürür.

Şebeke	Tanım	Örnek
Birli artı (`+`)	Tek operatör. Zaten değilse, işleneni bir sayıya dönüştürmeye çalışır .	`+"3"` döner `3` _ `+true` döner `1` _
Üs operatörü (`**`)	`base` Kuvveti hesaplar `exponent` , yani, `base^exponent`	`2 **` `3` döner `8` _ `10 ** -1` döner `0.1` _

bitsel operatörler

Bitsel bir işleç, işlenenlerini ondalık, onaltılık veya sekizlik sayılar yerine 32 bit (sıfırlar ve birler) kümesi olarak ele alır. Örneğin, dokuz ondalık sayının ikili gösterimi 1001'dir. Bitsel operatörler, işlemlerini bu tür ikili gösterimler üzerinde gerçekleştirir, ancak standart JavaScript sayısal değerlerini döndürürler.

Aşağıdaki tablo, JavaScript'in bitsel operatörlerini özetlemektedir.

Şebeke	kullanım	Tanım
bit düzeyinde VE	`a & b`	Her iki işlenenin karşılık gelen bitlerinin bir olduğu her bit konumunda bir bir döndürür.
bitsel VEYA	`a | b`	Her iki işlenenin karşılık gelen bitlerinin sıfır

Şebeke	kullanım	Tanım
		olduğu her bit konumunda bir sıfır döndürür.
bit düzeyinde XOR	`a ^ b`	Karşılık gelen bitlerin aynı olduğu her bit konumunda bir sıfır döndürür. [Karşılık gelen bitlerin farklı olduğu her bit konumunda bir tane döndürür.]
bit düzeyinde DEĞİL	`~ a`	İşleneninin bitlerini ters çevirir.
Sol shift	`a << b`	`a` İkili temsil `b` bitlerinde sola, sağdan sıfırlarda kayar .
İşaret yayma sağa kaydırma	`a >> b`	`a` İkili temsil `b` bitlerinde sağa kayar, kaydırılan bitler atılır .
Sıfır doldurma sağa kaydırma	`a >>> b`	`a` İkili temsil `b` bitlerinde sağa kayar , kaydırılan bitler atılır ve soldan sıfırlara kaydırılır.

Bit düzeyinde mantıksal işleçler

Kavramsal olarak, bitsel mantıksal işleçler şu şekilde çalışır:

- İşlenenler otuz iki bitlik tam sayılara dönüştürülür ve bir dizi bit (sıfırlar ve birler) ile ifade edilir. 32 bitten fazla olan sayıların en önemli bitleri atılır. Örneğin, 32 bitten fazla olan aşağıdaki tam sayı, 32 bitlik bir tam sayıya dönüştürülecektir:

```
Before: 1110 0110 1111 1010 0000 0000 0000 0110
0000 0000 0001
After:                  1010 0000 0000 0000 0110
0000 0000 0001
```

- Birinci işlenendeki her bit, ikinci işlenendeki karşılık gelen bit ile eşleştirilir: birinci bitten birinci bit'e, ikinci bitten ikinci bit'e vb.
- Operatör her bir bit çiftine uygulanır ve sonuç bit bazında oluşturulur.

Örneğin, dokuzun ikili gösterimi 1001 ve on beşin ikili gösterimi 1111'dir. Dolayısıyla bu değerlere bitsel operatörler uygulandığında sonuçlar aşağıdaki gibidir:

İfade	Sonuç	İkili Açıklama
`15 & 9`	`9`	`1111 & 1001 = 1001`
`15 | 9`	`15`	`1111 | 1001 = 1111`
`15 ^ 9`	`6`	`1111 ^ 1001 = 0110`
`~15`	`-16`	`~ 0000 0000 … 0000 1111 = 1111 1111 … 1111 0000`

İfade	Sonuç	İkili Açıklama
`~9`	`-10`	`~ 0000 0000 … 0000 1001 = 1111 1111 … 1111 0110`

32 bitin tamamının Bitwise NOT işleci kullanılarak ters çevrildiğini ve en önemli (en soldaki) bitin 1'e ayarlandığı değerlerin negatif sayıları (ikiye tümleyen temsili) temsil ettiğini unutmayın. `~x` değerlendiren aynı değere değerlendirir `-x - 1`.

Bitsel kaydırma işleçleri

Bitsel kaydırma işleçleri iki işlenen alır: birincisi kaydırılacak bir niceliktir ve ikincisi, birinci işlenenin kaydırılacağı bit konumlarının sayısını belirtir. Vites değiştirme işleminin yönü, kullanılan operatör tarafından kontrol edilir.

Kaydırma operatörleri, işlenenlerini otuz iki bitlik tamsayılara dönüştürür ve ya da türünden bir sonuç döndürür `Number` : `BigInt` özellikle, sol işlenenin türü ise `BigInt` , döndürürler `BigInt` ; aksi takdirde geri dönerler `Number` .

Kaydırma işleçleri aşağıdaki tabloda listelenmiştir.

Bitsel kaydırma işleçleri

Şebeke	Tanım	Örnek
Sola kaydırma (`<<`)	Bu operatör, ilk işleneni belirtilen bit sayısı kadar sola kaydırır. Sola	`9<<2` 36 verir, çünkü 1001 2 bit sola kaydırıldığında 100100 yani 36 olur.

Şebeke	Tanım	Örnek
	kaydırılan fazla bitler atılır. Sıfır bitler sağdan kaydırılır.	
İşareti ilerleten sağa kaydırma (>>)	Bu operatör, ilk işleneni belirtilen sayıda bit sağa kaydırır. Sağa kaydırılan fazla bitler atılır. En soldaki bitin kopyaları soldan içeri kaydırılır.	`9>>2` 2 verir, çünkü 1001 2 bit sağa kaydırıldığında 10 yani 2 olur. Aynı şekilde `-9>>2` işaret korunduğu için -3 verir.
Sıfır doldurma sağa kaydırma (>>>)	Bu operatör, ilk işleneni belirtilen sayıda bit sağa kaydırır. Sağa kaydırılan fazla bitler atılır. Sıfır bitler soldan kaydırılır.	`19>>>2` 4 verir, çünkü 10011 2 bit sağa kaydırıldığında 100 olur, bu da 4'tür. Negatif olmayan sayılar için, sıfır dolgulu sağa kaydırma ve işaret yayma sağa kaydırma aynı sonucu verir.

Mantıksal operatörler

Mantıksal işleçler genellikle Boolean (mantıksal) değerlerle kullanılır; olduklarında, bir Boole değeri döndürürler. Ancak, `&&` ve `||` işleçleri aslında belirtilen işlenenlerden birinin değerini döndürür, dolayısıyla bu işleçler Boole olmayan değerlerle kullanılırsa, Boolean olmayan bir değer döndürebilirler. Mantıksal işleçler aşağıdaki tabloda açıklanmıştır.

Mantıksal operatörler

Şebeke	kullanım	Tanım
Mantıksal VE (`&&`)	`expr1 && expr2`	`expr1` dönüştürülebiliyorsa döndürür `false` ; aksi takdirde, döner `expr2` . Bu nedenle, Boolean değerleriyle kullanıldığında, her iki işlenen de doğruysa `&&` döndürür ; `true` aksi takdirde, döner `false` .
Mantıksal VEYA (`||`)	`expr1 || expr2`	`expr1` dönüştürülebiliyorsa döndürür `true` ; aksi takdirde, döner `expr2` . Bu nedenle, Boolean değerleriyle kullanıldığında, işlenenlerden herhangi birinin doğru olup olmadığını `||` döndürür ; `true` her ikisi de yanlışsa, döndürür `false` .

Şebeke	kullanım	Tanım
Mantıksal DEĞİL (!)	`!expr`	Dönüştürülebilen `false` tek işleneni ise `true` ; aksi takdirde, döner `true` .

Dönüştürülebilecek ifade örnekleri, `false` null, 0, NaN, boş dize ("") veya tanımsız olarak değerlendirilen ifadelerdir.

Aşağıdaki kod, `&&` (mantıksal AND) operatörünün örneklerini gösterir.

```
const a1 =  true && true; // t && t returns true
const a2 =  true && false; // t && f returns false
const a3 = false && true; // f && t returns false
const a4 = false && (3 === 4); // f && f returns false
const a5 = 'Cat' && 'Dog'; // t && t returns Dog
const a6 = false && 'Cat'; // f && t returns false
const a7 = 'Cat' && false; // t && f returns false
```

Aşağıdaki kod, || (mantıksal VEYA) operatörü.

```
const o1 =  true || true; // t || t returns true
const o2 = false || true; // f || t returns true
const o3 =  true || false; // t || f returns true
const o4 = false || (3 === 4); // f || f returns false
const o5 = 'Cat' || 'Dog'; // t || t returns Cat
const o6 = false || 'Cat'; // f || t returns Cat
const o7 = 'Cat' || false; // t || f returns Cat
```

Aşağıdaki kod, ! (mantıksal DEĞİL) operatörü.

```
const n1 = !true; // !t returns false
const n2 = !false; // !f returns true
const n3 = !'Cat'; // !t returns false
```

Kısa devre değerlendirmesi

Mantıksal ifadeler soldan sağa değerlendirilirken, aşağıdaki kurallar kullanılarak olası "kısa devre" değerlendirmesi için test edilirler:

- `false && anything` kısa devre yanlış olarak değerlendirilir.
- `true || anything` kısa devre doğru olarak değerlendirilir.

Mantık kuralları, bu değerlendirmelerin her zaman doğru olduğunu garanti eder. *Yukarıdaki ifadelerin herhangi bir* kısmının değerlendirilmediğine dikkat edin , dolayısıyla bunu yapmanın herhangi bir yan etkisi etkili olmaz.

İkinci durum için, modern kodda, gibi çalışan Nullish birleştirme işlecini ( ) kullanabileceğinizi unutmayın , ancak yalnızca birincisi " nullish ", yani veya olduğunda ikinci ifadeyi döndürür . Bu nedenle, değerler ilk ifade için de benzer veya geçerli değerler olduğunda, varsayılanları sağlamak daha iyi bir alternatiftir . `??` `||` `null` `undefined` `''` `0`

BigInt operatörleri

Sayılar arasında kullanılabilen operatörlerin çoğu, `BigInt` değerler arasında da kullanılabilir.

```
// BigInt addition
const a = 1n + 2n; // 3n
// Division with BigInts round towards zero
const b = 1n / 2n; // 0n
// Bitwise operations with BigInts do not truncate
either side
```

```
const c = 4000000000000000n >> 2n; //
1000000000000000n
```

Bir istisna, BigInt değerleri için tanımlanmayan imzasız sağa kaydırmadır (`>>>`) . Bunun nedeni, bir BigInt'in sabit bir genişliğe sahip olmamasıdır, yani teknik olarak "en yüksek bit"e sahip değildir.

```
const d = 8n >>> 2n; // TypeError: BigInts have no
unsigned right shift, use >> instead
```

BigInts ve sayılar karşılıklı olarak değiştirilemez — hesaplamalarda bunları karıştıramazsınız.

```
const a = 1n + 2; // TypeError: Cannot mix BigInt
and other types
```

Bunun nedeni, BigInt'in sayıların bir alt kümesi veya

nedenle her iki taraftaki örtük dönüştürme, kesinliği kaybedebilir. İşlemin bir sayı işlemi mi yoksa BigInt işlemi mi olmasını istediğinizi belirtmek için açık dönüştürmeyi kullanın.

```
const a = Number(1n) + 2; // 3
const b = 1n + BigInt(2); // 3n
```

BigInts'i sayılarla karşılaştırabilirsiniz.

```
const a = 1n > 2; // false
const b = 3 > 2n; // true
```

Dize işleçleri

Dizi değerlerinde kullanılabilen karşılaştırma işleçlerine ek olarak, birleştirme işleci (+), iki dizi değerini bir araya getirerek iki işlenen dizisinin birleşimi olan başka bir dizi döndürür.

Örneğin,

```
console.log('my ' + 'string'); // console logs the
string "my string".
```

Kısa el atama operatörü, `+=` dizeleri birleştirmek için de kullanılabilir.

Örneğin,

```
let mystring = 'alpha';
mystring += 'bet'; // evaluates to "alphabet" and
assigns this value to mystring.
```

Koşullu (üçlü) işleç

Koşullu işleç, üç işlenen alan tek JavaScript işlecidir. Operatör, bir koşula bağlı olarak iki değerden birine sahip olabilir. sözdizimi şöyledir:

```
condition ? val1 : val2
```

true ise `condition` , operatörün değeri vardır `val1` . Aksi takdirde değeri vardır `val2` . Koşullu işleci, standart bir işleç kullandığınız her yerde kullanabilirsiniz.

Örneğin,

```
const status = age >= 18 ? 'adult' : 'minor';
```

`status` Bu ifade, if `age` onsekiz veya daha fazla ise değişkenine "yetişkin" değerini atar . Aksi takdirde, "minör" değerini atar `status` .

virgül operatörü

Virgül operatörü (`,`) her iki işlenenini de değerlendirir ve son işlenenin değerini döndürür. `for` Bu operatör , döngü boyunca her seferinde birden çok değişkenin güncellenmesine izin vermek için öncelikle bir döngü içinde kullanılır . Mecbur kalmadıkça başka yerde kullanmak üslup sayılır. Bunun yerine genellikle iki ayrı ifade kullanılabilir ve kullanılmalıdır.

Örneğin, `a` bir tarafında 10 öğe bulunan 2 boyutlu bir dizi ise, aşağıdaki kod iki değişkeni aynı anda güncellemek için virgül operatörünü kullanır. Kod, dizideki köşegen öğelerin değerlerini yazdırır:

```
const x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const a = [x, x, x, x, x];

for (let i = 0, j = 9; i <= j; i++, j--) {
//                                 ^
  console.log(`a[${i}][${j}]= ${a[i][j]}`);
}
```

tekli operatörler

Tekli işlem, yalnızca bir işlenenli bir işlemdir.

silmek

Operatör `delete` bir nesnenin özelliğini siler. sözdizimi şöyledir:

```
delete object.property;
delete object[propertyKey];
```

```
delete objectName[index];
```

burada `object` bir nesnenin adı, `property` var olan bir özellik ve `propertyKey` var olan bir özelliğe atıfta bulunan bir dizi veya sembol.

Operatör başarılı olursa `delete` , özelliği nesneden kaldırır. Daha sonra erişmeye çalışmak sonuç verecektir `undefined` . İşlem mümkün ise operatör `delete` geri döner ; işlem mümkün değilse `true` geri döner . `false`

```
delete Math.PI; // returns false (cannot delete non-
configurable properties)

const myObj = {h: 4};
delete myObj.h; // returns true (can delete user-
defined properties)
```

Dizi öğelerini silme

`delete` Diziler sadece nesneler olduğundan, teknik olarak onlardan elemanlar oluşturmak mümkündür . Ancak bu kötü bir uygulama olarak kabul edilir, bundan kaçınmaya çalışın. Bir dizi özelliğini sildiğinizde, dizi uzunluğu etkilenmez ve diğer öğeler yeniden dizine eklenmez. Bu davranışı elde etmek için, öğenin üzerine değeri yazmak çok daha iyidir `undefined` . Diziyi gerçekten değiştirmek için, gibi çeşitli dizi yöntemlerini kullanın `splice` .

bir çeşit

Operatör aşağıdaki yollardan biriyle kullanılır `typeof` :

```
typeof operand
```

Operatör `typeof` , değerlendirilmeyen işlenenin türünü gösteren bir dize döndürür. `operand` türün döndürüleceği dize, değişken, anahtar kelime veya nesnedir. Parantezler isteğe bağlıdır.

Aşağıdaki değişkenleri tanımladığınızı varsayalım:

```
const myFun = new Function('5 + 2');
const shape = 'round';
const size = 1;
const foo = ['Apple', 'Mango', 'Orange'];
const today = new Date();
```

Operatör `typeof` , bu değişkenler için aşağıdaki sonuçları döndürür:

```
typeof myFun;       // returns "function"
typeof shape;       // returns "string"
typeof size;        // returns "number"
typeof foo;         // returns "object"
typeof today;       // returns "object"
typeof doesntExist; // returns "undefined"
```

`true` ve anahtar sözcükleri için `null` operatör `typeof` aşağıdaki sonuçları verir:

```
typeof true; // returns "boolean"
typeof null; // returns "object"
```

Bir sayı veya dize için `typeof` operatör aşağıdaki sonuçları verir:

```
typeof 62;            // returns "number"
typeof 'Hello world'; // returns "string"
```

Özellik değerleri için `typeof` operatör, özelliğin içerdiği değer türünü döndürür:

```
typeof document.lastModified; // returns "string"
typeof window.length;         // returns "number"
typeof Math.LN2;              // returns "number"
```

Yöntemler ve işlevler için `typeof` operatör sonuçları aşağıdaki gibi döndürür:

```
typeof blur;        // returns "function"
typeof eval;        // returns "function"
typeof parseInt;    // returns "function"
typeof shape.split; // returns "function"
```

Önceden tanımlanmış nesneler için `typeof` operatör sonuçları aşağıdaki gibi döndürür:

```
typeof Date;     // returns "function"
typeof Function; // returns "function"
typeof Math;     // returns "object"
typeof Option;   // returns "function"
typeof String;   // returns "function"
```

geçersiz

Operatör aşağıdaki yollardan biriyle kullanılır `void`:

```
void (expression)
void expression
```

Operatör `void`, bir değer döndürmeden değerlendirilecek bir ifade belirtir. `expression` değerlendirmek için bir JavaScript ifadesidir. İfadeyi çevreleyen parantezler isteğe bağlıdır, ancak bunları kullanmak iyi bir stildir.

ilişki operatörleri

İlişkisel bir işleç, işlenenlerini karşılaştırır ve karşılaştırmanın doğru olup olmadığına bağlı olarak bir Boole değeri döndürür.

içinde

Belirtilen özellik belirtilen nesnedeyse işleç `in` geri döner . `true` sözdizimi şöyledir:

```
propNameOrNumber in objectName
```

burada `propNameOrNumber` bir özellik adını veya dizi dizinini temsil eden bir dize, sayısal veya sembol ifadesidir ve `objectName` bir nesnenin adıdır.

Aşağıdaki örnekler operatörün bazı kullanımlarını göstermektedir `in` .

```
// Arrays
const trees = ['redwood', 'bay', 'cedar', 'oak',
'maple'];
0 in trees;        // returns true
3 in trees;        // returns true
6 in trees;        // returns false
'bay' in trees;    // returns false (you must specify
the index number,
                   // not the value at that index)
'length' in trees; // returns true (length is an Array
property)

// built-in objects
'PI' in Math;          // returns true
const myString = new String('coral');
'length' in myString;  // returns true

// Custom objects
const mycar = { make: 'Honda', model: 'Accord', year:
1998 };
```

```
'make' in mycar;  // returns true
'model' in mycar; // returns true
```

örneği

Belirtilen nesne belirtilen nesne türündeyse işleç `instanceof` geri döner . `true` sözdizimi şöyledir:

```
objectName instanceof objectType
```

burada `objectName` karşılaştırılacak nesnenin adı `objectType` ve veya `objectType` gibi bir nesne türüdür . `Date` `Array`

`instanceof` Çalışma zamanında bir nesnenin türünü onaylamanız gerektiğinde kullanın . Örneğin, istisnaları yakalarken, atılan istisnanın türüne bağlı olarak farklı istisna işleme kodlarına dal verebilirsiniz.

Örneğin, bir nesne `instanceof` olup olmadığını belirlemek için aşağıdaki kod kullanılır . Bir nesne olduğu için , deyimdeki ifadeler yürütülür. `theDay Date theDay Date if`

```
const theDay = new Date(1995, 12, 17);
if (theDay instanceof Date) {
  // statements to execute
}
```

Temel ifadeler

Tüm işleçler sonunda bir veya daha fazla temel ifade üzerinde çalışır. Bu temel ifadeler, tanımlayıcıları ve sabit değerleri içerir , ancak birkaç başka tür de vardır. Aşağıda kısaca tanıtılmışlardır ve anlamları ilgili referans bölümlerinde ayrıntılı olarak açıklanmıştır.

Bu

Geçerli nesneye başvurmak için `this` anahtar sözcüğü kullanın . Genel olarak, `this` bir yöntemde çağıran nesneyi ifade eder. `this` Nokta veya parantez notasyonu ile kullanın :

```
this['propertyName']
this.propertyName
```

Adı verilen bir işlevin , nesne ve yüksek ve düşük değerler verildiğinde `validate` bir nesnenin özelliğini doğruladığını varsayalım: `value`

```
function validate(obj, lowval, hival) {
  if ((obj.value < lowval) || (obj.value > hival)) {
    console.log('Invalid Value!');
  }
}
```

`validate` Aşağıdaki örnekte olduğu gibi, her form öğesinin `onChange` olay işleyicisini `this` form öğesine iletmek için kullanarak arayabilirsiniz :

```
<p>Enter a number between 18 and 99:</p>
<input type="text" name="age" size="3"
onChange="validate(this, 18, 99);" />
```

Gruplandırma operatörü

Gruplandırma operatörü, `( )` ifadelerdeki değerlendirme önceliğini kontrol eder. Örneğin, önce toplamayı değerlendirmek için önce çarpma ve bölmeyi, ardından toplama ve çıkarma işlemlerini geçersiz kılabilirsiniz.

```
const a = 1;
const b = 2;
const c = 3;

// default precedence
a + b * c     // 7
// evaluated by default like this
a + (b * c)   // 7

// now overriding precedence
// addition before multiplication
(a + b) * c   // 9

// which is equivalent to
a * c + b * c // 9
```

yeni

Kullanıcı tanımlı bir nesne türünün veya yerleşik nesne türlerinden birinin örneğini oluşturmak için `new` işleci kullanabilirsiniz . `new` Aşağıdaki gibi kullanın :

```
const objectName = new objectType(param1, param2, /* …,
*/ paramN);
```

Süper

`super` Anahtar sözcük, bir nesnenin ebeveynindeki işlevleri çağırmak için kullanılır. Örneğin, sınıflarda üst yapıcıyı çağırmak yararlıdır .

```
super([arguments]); // calls the parent constructor.
super.functionOnParent([arguments]);
```

Son düzenleme: 23 Kasım 2022, MDN'ye katkıda bulunanlar tarafından