Operatörler

Çoğu operatörü okuldan hatırlarsınız. Toplama +, çarpma *, çıkarma - vs.

Bu bölümde okulda görmediğiniz aritmetiği işleyeceğiz.

1. Tanımlamalar: “unary”, “binary”, “operand”

Başlamadan önce terminolojiyi öğrenmekte fayda var.

- *Operand* operatörlerin uygulandığı(+,-,* vs.) değerlerdir. Örneğin çarpma işlemi için 5*2 örneğinden gidersek. İki tane operand vardır. Bunlardan solda olan 5 ve sağ operand 2. Bunlara argüman da denebilir.
- Eğer tek operanddan oluşursa bu operatör *unary* olarak adlandırılır. Örneğin, "-" sayının işaretini değiştirir:

```
let x = 1;

x = -x;

alert( x ); // -1, unary işlemi gerçekleşti
```

- Eğer operatörün iki tane operand’ı var ise buna **binary operand** denir. Örneğin çıkarma işlemi aşağıda bu formda bulunur.

```
let x = 1, y = 3;

alert( y - x ); // 2, iki sayının çıkarılması binary operand işlemidir.
```

Şeklen, iki operatörden konuşuyoruz. unary çıkarma (tek operand işareti değiştirir) ve binary çıkarma (iki operatör çıkarma)

Karakter dizisi birleştirme, binary +

JavaScript’te operatörlerin özel durumlarından birisi karakter dizilerinin + işareti ile birleştirilebilmesidir.

Böylece + işaretinin amacının ötesinde bir işlem yapabildiğinin farkına varmış olmalısınız.

Normalde + iki sayıyı toplamaya yaparken eğer bir taraf karakter dizisi ise bu durumda birleştirmeye yarar.

```
let s = "my" + "string";

alert(s); // mystring
```

Dikkat edin eğer iki operand’dan birisi karakter dizisi ise diğeri ne olursan olsun karakter dizisine çevrilir.

Örneğin:

```
alert( '1' + 2 ); // "12"

alert( 2 + '1' ); // "21"
```

Gördüğünüz gibi, ilk operand veya ikinci operandın karakter dizisi olması bir şeyi değiştirmiyor. Kural basit, her iki taraftan birisi karakter dizisi ise diğerini de karakter dizisine çevir ve birleştir.

Yani "+" işlemi hem birleştirme hem de tip değiştirme yapmaktadır. Bu sadece "+" operatörüne has bir olaydır.

Örneğin çıkarma ve çarpmanın davranışı farklıdır:

```
alert( 2 - '1' ); // 1

alert( '6' / '2' ); // 3
```

Sayısal değer dönüştürme, unary +

+ iki formda bulunur. Yukarıda kullandığımız binary form(iki tane operand olma olayı) veya unary form(tek operand olması).

Eğer unary + veya tek bir değerle kullanılan + işareti sayılar ile bir şey yapmaz. Fakat eğer bu bir sayı değilse sayıya çevrilir.

Örneğin:

```
// Sayılara bir etkisi yoktur

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// Sayı olmayan değerleri çevirir

alert( +true ); // 1

alert( +"" );   // 0
```

Aslında Number(...) işlemini yapar. Fakat daha kısa biçimyle.

Karakter dizilerini sayılara çevirme gerekliliği sıklıkla önünüze gelir. Örneğin HTML form değerlerini alırken sadece karakter dizisi kullanır. Fakat ya siz bunları toplamak istiyorsanız ?

Bildiğiniz gibi iki karakter dizisini + işareti ile toplarsanız birleştirme işlemi yapar:

```
let elma = "2";

let portakal = "3";

alert( elma + portakal ); // "23",  binary toplama iki karakter dizisini birleştiriyor
```

Eğer sayı olarak kullanmak istiyorsanız, önce dönüştürme işlemini yapıp sonra toplayabilirsiniz.

```
let elma = "2";

let portakal = "3";

// her iki değer de binary toplama işleminden önce sayıya çevrilmişlerdi

alert( +elma + +portakal ); // 5

// Daha uzun bi şekilde bu işlemi yapmak istiyorsanız

// alert( Number(apples) + Number(oranges) ); // 5
```

// şeklinde yapabilirsiniz.

Olaya bir matematikçi gözünden bakarsanız + kullanımı garip gelebilir. Fakat bir programcının gözünden özel bir olay yok aslında: operand'ı bir tane olan(unary) toplama işlemi önce uygulanıyor ve karakter dizisini sayıya çeviriyor. Daha sonra iki tane operand'lı (binary) toplama işlemi bunları topluyor.

Neden önce "unary" işlemi gerçekleşiyor da "binary" işlemi gerçekleşmiyor? Buna *yüksek öncelik* diyebiliriz.

2. Operatör Öncelikleri

Eğer bir ifade birden fazla operatör içeriyorsa. Bu ifade çalıştırılırken tanımlı *önceliklere* göre çalıştırılır, bir başka ifade ile öncelik sırasına göre çalıştırılır.

Okuldan hepinizin hatırlayacağı gibi çarpma işlemi toplamadan önce yapılır 1 + 2 * 2. Aslında *öncelik* tam olarakta budur. Çarpma işlemi toplama işleminden daha *yüksek önceliğe* sahiptir.

Parantez, bu öncelikleri çiğner ve eğer bu *önceliklerden* memnun değilseniz bunları tekrar tanımlamanıza olanak verir. Örneğin (1 + 2) * 2

JavaScript dilinde birçok operatör vardır. Her operatörün de bir önceliği. Yüksek öncelik sayısına sahip operatör önce çalışır. Eğer öncelik değerleri eşit ise soldan sağa doğru çalışır.

Öncelik tablosu (Ezberlemenize gerek yok sadece unary operatörlerin binary olanlara göre daha üstün olduğunu hatırlayın yeter). Yani +elma + +portakal işleminde önce unary ile elma'nın değerini sayı yapar sonra portakal'ın değerini sayı yapar ve en sonunda toplar.

Öncelik	Adı	İşareti
…	…	…
16	unary toplama	+
16	unary çıkarma	-
14	çarpma	*
14	bölme	/
13	toplama	+
13	çıkarma	-
…	…	…
3	atama	=
…	…	…

Görüleceği üzere "unary toplama", 16 ile normal toplama işlemi (13- binary toplama) nin öncesindedir.

3. Atama

Atama operatörü = dir. Öncelik sırasında en altlarda yer almaktadır. Böylece x = 2 * 2 + 1 ifadesi çalıştığında önce tüm işlemler yapılır ardından "=" çalıştırılarak sonuç x içerisinde tutulur.

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

Zincirleme atama yapmak şu şekilde mümkündür:

```
let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4
```

Zincirleme atama sağdan sola doğru olur. Önce en sağdaki değişkene değer atanır. 2+2 değeri önce c'ye ardından b ve son olarak da a'ya atanır. En sonunda tüm değişkenler tek bir değeri alırlar.

"=" operatörü değer döndürür

Operatör her zaman değer döndürür. Toplama + veya çarpma için * bu çok açıktır. Fakat ya atama ? Atama operatörü de aslında değer döndürür.

Aşağıdaki gibi bir işlem yaptığınızda value x'in içine yazılır ve sonra döndürülür.

Daha karmaşık bir örnek şu şekilde yapılabilir:

```
let a = 1;

let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3

alert( c ); // 0
```

Yukarıdaki örnekte, (a = b+1) in sonucu a ya atandıktan sonra(3) 3'den çıkarmak için kullanılıyor.

Komik bi kod değil mi? Nasıl çalıştığını anlamanız lazım, bazen başka kütüphaneler kullandığınızda böyle şeyleri sizin yazmanız beklenmez. Böyle olaylar aslında kodun okunaklılığını azaltır.

Kalan: %

Kalan % operatörü yüzde ile alakası olmayan bir operatördür.

a % b a'nın b'ye bölümünden kalan değeri verir.

Örneğin:

```
alert( 5 % 2 ); // 5'in 2 ile bölümünden kalan 1'dir.
```

```
alert( 8 % 3 ); // 8'in 3 ile bölümünden kalan 2'dir.

alert( 6 % 3 ); // 6'nın 3 ile bölümünden kalan 0'dır.
```

Üs alma **

Üs alma operatörü JavaScript diline sonradan eklenen bir operatördür.

Doğal sayı olan b değeri için a ** b a'nın b defa kendisiyle çarpılması demektir.

Örneğin:

```
alert( 2 ** 2 ); // 4  (2 * 2)

alert( 2 ** 3 ); // 8  (2 * 2 * 2)

alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```

Integer olmayan değerler için de aynı işlemi yapmak mümkün örneğin:

```
alert( 4 ** (1/2) ); // 2 ( 1/2 üstü karekökü anlamına da gelir.)

alert( 8 ** (1/3) ); // 2 (1/3 üstü ise küp kök anlamına gelir. )
```

Artırma/Azaltma

Bir sayıyı artırmak veya azlatmak sayısal operasyonlarda önemli sayılabilecek bir düzeydedir.

Bunun için özel bir operatör yapılmıştır:

- **Artırma** ++ değişkenin değerini 1 artırır:
- let sayac = 2;
- sayac++; // sayac = sayac + 1 ile aynı, fakat daha kısa

```
alert( sayac ); // 3
```

- **Azaltma** -- değişkenin değerini bir azaltır:
- let sayac = 2;
- sayac--; // sayac = sayac - 1 ile aynı, fakat daha kısa

```
alert( sayac ); // 1
```

Önemli:

Artırma/Azaltma sadece değişkenlere uygulanabilirler. 5++ gibi bir kullanım hata verecektir.

++ ve -- operatörleri değişkenden önce veya sonra kullanılabilirler.

- Operatör değişkenden sonra geliyorsa ona “postfix form” deriz: counter++.
- “prefix form” ise opeatörün değişkenden önce geldiği durumdur: ++counter.

Bu iki durumda da aynı işlem yapılır: counter değişkeni 1 arttırılır.

Peki bir farkları var mı? Evet, fakat bunu ++/-- işleminden dönen değerleri kullanırsak görebiliriz.

Şöyle açıklayabiliriz. Bildiğimiz üzere tüm operatörler bir değer döndürür. arttırma/azaltma opeatörleri buna bir istisna değildir. Prefix formu oluşan yeni değeri döndürürken, postfix formu eski değeri(arttırma/azaltma işlemi yapılmadan önceki) döndürür.

Farkı görebilmemiz için örneği inceleyelim:

```
let counter = 1;

let a = ++counter; // (*)

alert(a); // 2
```

(*) satırında *prefix* formundaki ++counter counter değişkenini arttırır ve yeni değer olan 2 yi döndürür. Yani alert bize 2 değerini gösterecektir.

Şimdi de postfix kullanıma bakalım:

```
let counter = 1;

let a = counter++; // (*) changed ++counter to counter++

alert(a); // 1
```

(*) satırında *postfix* formundaki ++counter de aynı şekilde counter değişkenini arttırır fakat bu sefer değişkenin *eski* değerini(arttırma işlemi yapılmadan önceki) değerini döndürür.
Yani alert bize 1 değerini gösterecektir.

Özetle:

- Eğer arttırma/azaltma işleminin sonucunu kullanmıyorsak hangi formu kullandığımızın bir farkı olmaz:
- let counter = 0;
- counter++;
- ++counter;

alert(counter); // 2, iki satır da aynı işlemi yaptı.

- Eğer bir değeri arttıracak ve onu aynı anda(o işlem sırasında) kullanacaksak, prefix formunu kullanmamız gerekir:
- let counter = 0;

```
alert( ++counter ); // 1
```

- Eğer arttırma yapacak fakat arttırma yapmadan yapmadan önceki değeri kullanacaksak, postfix formunu kullanmamız gerekir:
- let counter = 0;

```
alert( counter++ ); // 0
```

Diğer operatörler arasında arttırma/azaltma

++/-- operatörleri ayrıca bir ifadenin içinde kullanılabilirler. Öncelikleri diğer tüm operatörlerden daha yüksektir.

Örneğin:

```
let counter = 1;

alert( 2 * ++counter ); // 4
```

alttaki örnek ile karşılaştıralım:

```
let counter = 1;

alert( 2 * counter++ ); // 2, çünkü counter++ "eski" değeri döndürecektir
```

Teknik olarak doğru olmakla birlikte bu tür kullanımlar kodu daha az okunur kılar. Bir satırında birden çok işlem yapılması çok iyi değildir.

Kod okurken hızlı bir göz taraması sırasında counter++ ifadesini gözden kaçırmamız oldukça olasıdır. Değişkenin arttırıldığı açıkça gözükmeyebilir.

"Bir satır – bir işlem" stili önerilir:

```
let counter = 1;

alert( 2 * counter );

counter++;
```

4. Bitsel(Bitwise) Operatörler

Bitsel operatörler argümanlara 32-bitlik doğal sayı gibi davranır ve ikili gösterimleri düzeyinde çalışır.

Bu operatörler JavaScript'e özgü değildir. Çoğu programlama dilinde bulunurlar.

Operatörlerin listesi:

- AND – VE (&)
- OR – VEYA (|)
- XOR – ÖZEL VEYA (^)
- NOT – DEĞİL (~)
- LEFT SHIFT – SOLA KAYDIRMA (<<)
- RIGHT SHIFT – SAĞ KAYDIRMA (>>)
- ZERO-FILL RIGHT SHIFT – SIFIR DOLDURARAK SAĞ KAYDIRMA (>>>)

Bu oparatörler çok nadir kullanılır. Onları anlamak için düşük seviyeli sayı temsiline girmemiz gerekiyor ve özellikle de yakın zamanda onlara ihtiyaç duymayacağımızdan şu anda bunu yapmak uygun olmayacaktır. Merak ediyorsanız, MDN ile ilgili Bitwise Operators makalesini okuyabilirsiniz. Gerçekten ihtiyacınız olduğunda bunu yapmak daha doğru olacaktır.

Modify-in-place (Yerinde Değiştir)

Bazen bir değişken üzerinde bir operatör işlemi yaparız ve yeni oluşacak değerini aynı değişkende tutmak isteriz.

Örneğin:

```
let n = 2;

n = n + 5;

n = n * 2;
```

Bu işlemler += ve *= kullanılarak kısaltılabilir:

```
let n = 2;

n += 5; // şu an n = 7 (n = n + 5 ile aynı)

n *= 2; // şu an n = 14 (n = n * 2 ile aynı)

alert( n ); // 14
```

Kısa olan "modify-and-assign" operatörleri tüm aritmetik ve bitsel operatörler için mevcuttur: /=, -=, vb.

Bu tür operatörler normal bir atama(assignment) ile aynı önceliğe sahiptir, bu yüzden diğer birçok hesaplamalardan sonra çalışırlar.

```
let n = 2;

n *= 3 + 5;


alert( n ); // 16  (önce sağ kısımda işlem yapıldı, n *= 8 gibi)
```

5. Virgül

Virgül operatörü , nadir ve en alışılmadık operatörlerden birisidir. Bazen daha kısa kodlar yazmak için kullanılır. Bu yüzden neler olduğunu anlamak için bu operatörü de bilmemiz gerekiyor.

Virgül operatörü birden fazla ifadeyi virgül , ile ayırarak hesaplamamıza olanak sağlar. Her bir ifade işleme alınır fakat bu ifadelerden sadece sonuncusu döndürülür.

Örneğin:

```
let a = (1 + 2, 3 + 4);

alert( a ); // 7 (3 + 4 işleminin sonucu)
```

Burada, ilk ifade olan 1 + 2 işleme giriyor fakat sonucu çöpe atılıyor. Sonrasında gelen 3 + 4 işleme giriyor ve sonuç olarak geri döndürülüyor.

Virgül operatörünün önceliği çok düşüktür

Unutmamak gerekir ki; virgül operatörü çok düşük bir önceliğe sahiptir, önceliği ='den bile daha düşüktür. Bu yüzden yukarıdaki örnekte gördüğümüz gibi parantezler çok önemlidir.

Parantezler olmadan: a = 1 + 2, 3 + 4 ifadesinde önce + işleme alınır, değerler toplanarak a = 3, 7 ifadesine çevirilir, ondan sonra atama operatörü = ile a = 7 ataması yapılır, ve sonuç olarak virgülden önceki sayı olan 3 işlenmeyerek yok sayılır.

Peki neden son kısım hariç her şeyi yok sayan bir operatöre ihtiyacımız var?

Bazen bizler; bir satırda birkaç işlem yapılan karmaşık yapılarda bu operatörü kullanırız.

Örneğin:

```
// Bir satırda 3 farklı işlem

for (a = 1, b = 3, c = a * b; a < 10; a++) {

 ...

}
```

Bu tarz numaralar birçok JavaScript frameworklerinde kullanılır. Bu yüzden bunlardan bahsettik. Ama genelde bunlar kodun okunabilirliğini azaltıyorlar. Bu yüzden kullanmadan önce iyi düşünmek gerekir.

Görevler

Önden ve sonradan eklemeli değişkenler.

önem: 1

Aşağıdaki işlemlerden sonraki a, b, c,d değerlerini yazınız?

```
let a = 1, b = 1;


let c = ++a; // ?

let d = b++; // ?
```

Atama sonuçları

önem: 2

Aşağıdaki işlemler sonrasında a ve x değerleri nedir ?

```
let a = 2;

let x = 1 + (a *= 2);
```