**Java Programcıları için ECMAScript (Javascript)**

**Oğuz KARAN**

Javascript ilk olarak Netscape firması tarafından geliştirilmiş ve ilk olarak “Livescript” ismi verilmiştir. 1997 yılında Ecma tarafından standartlaştırılmış ve “EcmaScript” ismini almıştır. Ecmascript Ecma-262 dökümanı ile standartlaştırlmıştır. Ecmascript aslında bir programlama dilidir. En çok Web ortamında kullanıldığından birçok browser tarafından neredeyse standart kabul edilen teknolojiler eklenmiştir.

**Ecmascript Programlama Dili**

Ecmascript yorumlayıcı ile çalışan bir dildir. Herhangi bir derleme işlemi olmadığından aşağı seviyeli bir kod üretilmez. Ancak bazı yorumlayıcılar hızlı çalışma açısından ara kodlar üretebilir. Ancak programcı için ES yorumlayıcı ile çalışan bir dildir. Bu durum programcının yaptığı hataların hemen görülememesine sebep olabilir. Ecmascript çıktığından beri en çok web ortamında ön yüzde kullanılmaktadır. Ancak son yıllarda geliştirilen teknolojilerle server tarafta çalışan uygulamalar da ecmascript kullanılarak geliştirilebilmektedir. Javascript ayrıca mobil geliştirmede de oldukça yoğun olarak kullanılmaktadır.

**Türler:**

Türler şu şekildedir:

- number

- string

- boolean

- null

- undefined

- Symbol

- function

number türü “IEEE 754 double precision” formatına göre sayıların tutulduğu bir türdür. Tamsayı ya da gerçek sayı ayrımı yapılmamıştır. Tüm sayılar bu formatta tutulur.

string bir değer türü olarak alınmıştır. Tüm dillerde olduğu gibi yazı işlemlerinde kullanılmaktadır.

boolean türü klasik olarak mantıksal işlemler için (doğru veya yanlış) düşünülmüştür.

null türü özellikle referans türleri için düşünülmüştür ve hiç bir nesneyi göstermeyen adres değeridir.

undefined henüz değer almamış bir değişkenin türüdür.

Symbol türü ES6 ile eklenmiş ve sembol tanımlamakta kullanılan bir türdür.

function türü bir fonksiyonu temsil eden türdür. İçerisine fonksiyon adresi atanmış değişkenler function türündendir.

number, string ve boolean türlerini sarmalayan (wrapper) sırasıyla Number, String ve Boolean isimli nesneler (object) bulunmaktadır.

**ECMAScript'de Değişkenler**

ECMAScript'de genel olarak 4(dört) yerde değişken bildirimi yapılabilir.

1. Global olarak dosya içerisinde tüm nesne, sınıf ve fonksiyonların dışında

2. Yerel değişken olarak bloklar içerisinde

3. Fonksiyonların veya metotların parametre değişkenleri olarak

4. Nesnelerin ve sınıfların veri elemanları olarak

Global değişkenler içerisinde değer atanarak doğrudan isimleriyle bildirilebilirler. Bu durumda bildirildikleri yerden o dosyanın sonuna kadar görülebilir olurlar:

**a** = 10  
  
***console***.log(**a**)

Doğrudan bildirilen değişkenlerin nerede bildirildiğinin önemi yoktur. Yani bir fonksiyon içerisinde bildirilseler bile global değişken olurlar. Global değişkenler var anahtar sözcüğü ile de bildirilebilir.

Global değişkenler içerisine değer atanmadan bildirilirse undefined değerini alırlar. Şüphesiz bu durumda değişkenin var ile bildirilmesi gerekir:

**var a**  
  
***console***.log(**a**)

***Anahtar notlar:*** *ES6 ile birlikte let anahtar sözcüğü ile de değişken bildirimi yapılabilmektedir.*

ES'de değişkenlerin içerisinde tutulan değerlerin türleri değişebilir:

**a** = **"ankara"** *//string***a** = 10 *//number*

***Anahtar notlar:*** *Yukarıdaki tanım daha iyi açıklamak içindir. Aşağı seviyede bu şekilde olmaz. Bu durum ileride ele alınacaktır.*

Bir değişkenin yerel olması için kesinlikle var ya da let anahtar sözcükleri ile bildirilmesi gerekir:

**function** *main*()  
{  
 **var** a = **"ankara"**;  
  
 *foo*();  
}  
  
**function** *foo*()  
{  
 ***console***.log(a) // error  
}  
  
*main*()

Bu cümlenin tersi her zaman doğru değildir. Yani her var veya let olarak bildirilen değişken yerel olmayabilir.

Nesnenin ve sınıfın veri elemanları daha sonra ele alınacaktır.

**let ve var Anahtar Sözcükleri Arasındaki Farklar**

let ile bildirilen yerel değişkenler ile var ile bildirilen yerel değişkenler arasındaki fark, let yerel değişkenler blok faaliyet alanına sahiptir, var değişkenler fonksiyon faaliyet alanına sahiptir. Yani var değişkenler bir fonksiyon içerisinde bildirilmişlerse içteki blokta bile olsalar tüm fonksiyon boyunca görülebilirler. let değişkenler ise bildirildiği yerden bildirildiği bloğun sonuna kadar görülebilirler:

***function main()  
{  
 foo();  
}  
  
function foo()  
{  
 {  
 let a = 10  
 }  
  
 console.log(a) //error  
}  
  
main()***

***function main()  
{  
 foo();  
}  
  
function foo()  
{  
 {  
 var a = 10  
 }  
  
 console.log(a)  
}  
  
main()***

let ve var değişkenlerin global olması durumunda faaliyet alanı bakımından farkları yoktur. ES’de mümkün olduğunca let kullanılmalıdr. Doğrudan bildirilen değişkenlerin ise hiç kullanılmaması tavsiye edilir.

Aşağıdaki örneği inceleyiniz:

***function main()  
{  
 foo();  
}  
  
function foo()  
{  
 {  
 let a = 20   
 var a = 10 //error  
  
 }  
  
  
 console.log(a)  
}  
  
main()***

Yukarıdaki örnekte let yerine var kullanılsaydı bir error oluşmazdı:

***function main()  
{  
 foo();  
}  
  
function foo()  
{  
 {  
 var a = 20  
 var a = 10  
 }  
  
  
 var a = 30  
 console.log(a)  
}  
  
main()***

Peki sizce bu kod ne kadar anlamlı?

**Değişken İsimlendirme Kuralları**

ES’ de değişken isimlendirme kuralları Java programlama dili ile aynıdır.

***Anahtar Notlar:*** *Alttire karakterinin Java 9 ile birlikte tek başına değişken ismi olması yasaklanmıştır. Ancak ES'de bu geçerlidir.*

**Global Fonksiyonlar**

Global fonksiyonlar dosya içerisinde hiç bir nesneye veya sınıfa ait olmayan ve yerel olarak bildirilmeyen fonksiyonlardır. Fonksiyon bildirimleri aynı zamanda tanımlamadır.

Global fonksiyonlar function anahtar sözcüğü ile bildirilirler.

**function** *foo*()  
{  
 ***console***.log(**"foo"**)  
}

Fonksiyonlar parametrik yapıları nasıl olursa olsun istenildiği kadar argüman ile çağrılabilirler:

*foo*()  
*foo*(20)  
*foo*(10, 20, 30)

Fonksiyonlar geri dönüş değeri olmasa da geri dönüş değeri varmış gibi kullanılabilir. Bu durumda geri dönüş değeri olarak undefined değeri elde edilecektir.

**function** *foo*()  
{  
 ***console***.log(**"foo"**)  
}

let ***a*** = *foo*(20)  
  
***console***.log(***a***)

Geri dönüş değeri için yine return deyimi kullanılır. Return deyiminin tek başına kullanılması durumunda fonksiyondan çıkılır ve değer döndürülmez (aslında undefined döndürülmüş olarak düşünülebilir):

**function** *foo*(val)  
{  
 **if** (val > 0)  
 **return  
  
 return** val \* val  
}  
  
**let *a*** = *foo*(-20)  
  
***console***.log(***a***)

Yukarıdaki örnekte fonksiyon akış durumuna göre undefined ya da argümanın karesine döner. ES kullanan programcının "her fonksiyonun zaten bir geri dönüş değeri var içeride yazılan koda göre geri dönüş değeri anlamlanacak" bakış açısıyla doğru şekilde düşünerek kodu yazması gerekir. Yani fonksiyonun geri dönüş değeri olacaksa akışın her noktasında geri dönülmesi anlamlıdır. Fonksiyonun geri dönüş değeri olmayacaksa hiç bir şekilde return, ifade ile döndürülmemelidir.

ES'de bir fonksiyon parametre değişkeni alabilir. Bu durumda parametre değişkenlerinin yalnızca ismi yazılır:

**function** *square*(val)  
{  
 **return** val \* val  
}  
  
**let *res*** = *square*(-20)  
  
***console***.log(***res***)

Fonksiyonlara bakıldığında parametrelerin türlerinin anlaşılması her zaman mümkün olmayabilir. Bu durumda iki yaklaşım kullanılabilir:

1. Fonksiyonlar yazılırken parametreler için herhangi bir tür kontrolü yapılmaz. İyi bir dökümantasyonla kullanan programcıya bilgi verilir. Yanlış argüman geçilmesinden programcı sorumludur. Örneğin:

**function** *square*(val)  
{  
 **return** val \* val  
}  
  
**let *res*** = *square*(**"ankara"**)  
  
***console***.log(***res***)

Yukarıdaki kodda square fonksiyonuna geçilen argüman string türünden olduğu için NaN değeri elde edilir. Bu da fonksiyonu çağıranın sorumluluğundadır.

2. Fonksiyonların parametre değişkenlerinin türleri çağrılma sırasında kontrol edilebilir. Bu yaklaşım sürekli kontrol gerektirdiğinden çalışmayı yavaşlatabilir. Örneğin:

**function** *square*(val)  
{  
 **if** (**typeof**(val) != **"number"**)  
 **throw *Error***(**"unsupported argument"**)

**return** val \* val  
}  
  
**let *res*** = *square*(**"ankara"**)  
  
***console***.log(***res***)

Yukarıdaki örnekte istenen türden argüman için de kontrol yapılır. Bu da iyi bir teknik olmayabilir.

Fonksiyon yazımında birinci yaklaşım çok daha fazla kullanılmaktadır.

ES'de function overloading kavramı yoktur. Aynı isimde iki tane fonksiyon yazılırsa yukarıdan aşağıya son yazılanı çağrılırken geçerli kabul eder:

**function** *foo*()  
{  
 ***console***.log(**"foo1"**)  
}  
  
*foo*()  
  
**function** *foo*()  
{  
 ***console***.log(**"foo2"**)  
}  
  
*foo*()

Fonksiyonlar parametre sayısı önemli olmaksızın teorik olarak istenildiği sayıda argümanla çağrılabilirler. Bu durumda programcı fonksiyon içerisinde isterse argümanları kontrol edebilir:

**function** *main*()  
{  
 **let** result = *sum*(10, 20, 30)  
  
 ***console***.log(result)  
  
 result = *sum*(10, 20)  
  
 ***console***.log(result)  
}  
  
**function** *sum*()  
{  
 **let** result = 0;  
  
 **for** (**let** index **in *arguments***)  
 result += ***arguments***[index];  
  
 **return** result;  
}  
  
*main*()

Yukarıda, her metot içerisine iliştirilmiş olan arguments dizisi kullanılmıştır. Bu konu ileride ele alınacaktır.

**Sabitler**

Program içerisinde doğrudan yazılan değerlere sabit denilmektedir. Sabitler ES içerisinde çeşitli kategorilere ayrılmaktadır. Sabitler sayı biçimindeyse nokta içersin ya da içermesin “number” türündendir. Ayrıca boolean türden *true* ve *false* isimli sabitler de bulunmaktadır. Standartlara göre string atomları da sabit (string literal) olarak ele alınmaktadır. Sayısal sabitlerin çeşitli sayı sistemlerinde gösterilişi vardır:

**function** *main*()  
{  
 **let** a = 0xA  
  
 ***console***.log(a)  
  
 a = 0o12  
  
 ***console***.log(a)  
  
 a = 10  
  
 ***console***.log(a)  
  
 a = 0b1010  
  
 ***console***.log(a)  
}  
  
  
*main*()

ES programlama dilinde karakter sabiti yoktur. Tek tırnak içerisindeki atomlar da string türündendir. Ayrıca önceden tanımlanmış ters bölü sabitleri de (escape sequence) ES' de geçerlidir. Escape sequence karakterlerin listesi şu şekildedir:

|  |  |
| --- | --- |
| \b | Backspace |
| \t | Tab |
| \n | Line feed |
| \v | Vertical Tab |
| \f | Form feed |
| \r | Carriage return |
| \” | Double quote |
| \’ | Single quote |
| \\ | Back slash |
| \` | Backtic |

***Anahtar Notlar:*** *Aşağıda kullanılan write ve writeln fonksiyonlarının nasıl yazıldıkları şu an için önemsizdir. write ve writeln “standard output (stdout)“’ yazma yapalar ve sırasıyla cursor’ı son yazılan karakterin sonuna, ve alt satırın başına indirirler (line feed)*

**Temel Operatörler**

Belirli bir işleme yol açan ve bir değer üreten atomlara operatör denir. Operatörler sınıflandırılabilir:

**Aritmetik Operatörler**

Aritmetik operatörler klasik 4(dört) işlem operatörleri ile işaret + ve – operatörleri ve ++ ve -- operatörleridir. Bu operatörlerin kullanımı genel olarak Java ile aynıdır. Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = 10  
 **let** b = a++  
 **let** c = ++a  
  
 *writeln*(a) *//12  
 writeln*(b) *//10  
 writeln*(c) *//12*}  
  
*main*()

Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *digitsSum3Digits*(val)  
{  
 let a = *parseInt*(val / 100)  
 let b = *parseInt*(val / 10) % 10  
 let c = val % 10  
  
 return Math.abs(a + b + c)  
}  
  
function *main*()  
{  
 *writeln*(*digitsSum3Digits* (123))  
 *writeln*(*digitsSum3Digits* (-345))  
 *writeln*(*digitsSum3Digits* (-123))  
 *writeln*(*digitsSum3Digits* (200))  
}  
  
*main*()

***Anahtar Notları:*** *ES’de tamsayı türü olmadığından yapılan her işlem de gerçek sayılara göre yapılır. Bazen bir gerçek sayının noktadan sonraki kısmının atılıp yalnızca tam kısmın elde edilmesi gerekebilir. Bu durumda ES’ de global bir fonksiyon olan parseInt kullanılabilir. Bu fonksiyonun detayları ileride ele alınacaktır.*

**Karşılaştırma Operatörleri**

Temel karşılaştırma operatörleri 8 tanedir: ==, !=, ===, !==, >, <, >=, <=

Burada eşitlik testi yapan operatörler birçok programlama dilinden farklıdır.

== ve != operatörleri soyut karşılaştırma (abstract comparison) yaparlar. == ve != operatörleri kullanılırken işlem öncesi otomatik tür dönüştürme kuralları uygulanır. Bu kurallar ileride ele alınacaktır.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = **"10"**;  
 **let** b = 10;  
  
 *writeln*(a == b);  
}  
  
  
*main*()

Bu operatörler kullanılırken dikkatli olunmalıdır. Tam anlamıyla eşitlik karşılaştırması (strictly equal or strictly not equal) === ve !== operatörleriyle yapılabilir. Bu operatörler karşılaştırılan türler birbirinden farklıysa kesinlikle false değerini üretirler:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = **"10"**;  
 **let** b = 10;  
  
 *writeln*(a === b);  
}  
  
  
*main*()

Programcı öncelikle bu operatörleri düşünmeli, soyut eşitlik karşılaştırma operatörleri gerektiğinde kullanılmalıdır. Ancak === ve !== operatörlerinin fazladan kontrol yaptığı da unutulmamalıdır. Soyut eşitlik karşılaştırma operatörlerinin kullanılması zorunluluğu çok azdır.

**Mantıksal Operatörler**

Mantıksal operatörlerden and ve or operatörleri klasik kısa devre davranışına sahiptir. Bu operatörler ifadenin değerini öncelik sırasıyla yapıldığında elde edilecek sonuca ulaşacak şekilde en kısa yoldan hesaplarlar. Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** result = *foo*() || *bar*() && *tar*()  
  
 *writeln*(result)  
}  
  
**function** *foo*()  
{  
 *writeln*(**"foo"**)  
 **return true**  
}  
  
**function** *bar*()  
{  
 *writeln*(**"bar"**)  
 **return false**  
}  
**function** *tar*()  
{  
 *writeln*(**"tar"**)  
 **return true**  
}  
  
  
*main*()

Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let result = *bar*() && *foo*() || *tar*()  
  
 *writeln*(result)  
}  
  
function *foo*()  
{  
 *writeln*("foo")  
 return true  
}  
  
function *bar*()  
{  
 *writeln*("bar")  
 return false  
}

function *tar*()  
{  
 *writeln*("tar")  
 return false  
}  
  
  
*main*()

Bitsel “and” ve “or” operatörleri boolean türü ile işleme sokulduğunda kısa devre özelliği olmayan “and“ ve “or” operatörleri gibi davranırlar:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** result = *foo*() | *bar*() & *tar*()  
  
 *writeln*(result)  
}  
  
**function** *foo*()  
{  
 *writeln*(**"foo"**)  
 **return true**;  
}  
  
**function** *bar*()  
{  
 *writeln*(**"bar"**)  
 **return false**;  
}

**function** *tar*()  
{  
 *writeln*(**"tar"**)  
 **return true**  
}  
  
*main*()

Burada elde edilen sonuç number türündendir.

Aslında bitsel operatörler numeric türler için bitsel işlemlerde kullanılır. Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = 10  
 **let** b = 11  
 **let** c = a & b  
  
 *writeln*(**`c=**${c}**`**)  
  
 c = a | b  
 *writeln*(**`c=**${c}**`**)  
}  
  
*main*()

**Atama Operatörü**

Atama operatörü iki operandlı bir operatördür. Atama işleminde birinci operandın türü atanan ifadenin türüne göre değişebilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = 10  
 **let** b = **true**  
  
 *writeln*(**typeof**(a))  
 *writeln*(**typeof**(b))  
  
 a = b  
  
 *writeln*(**typeof**(a))  
 *writeln*(**typeof**(b))  
}  
  
*main*()

ES'de ismine işlemli atama denilen (augmented assignment) bir grup operatör bulunmaktadır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = 10;  
 **let** s = **"11"**

a += s; *// a = a + s  
 writeln*(a)  
}  
  
*main*()

İşlemli atama operatörleri kodun yazımını ve okunabililiğini basitleştirir. Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = 10  
 **let** b = 20  
 **let** c = 2  
  
 a \*= b + c *//a = a \* (b + c)  
   
 writeln*(a)  
}  
  
*main*()

**Virgül Operatörü**

Bu operatör iki operandlı (binary) araek (infix) durumunda bir operatördür. Virgül operatörüne ilişkin ifadede önce birinci operandına ilişkin ifade, sonra da ikinci operandına ilişkin ifade yapılır. Operatör öncelik tablosunun en sonunda olan operatördür. Ürettiği değer ikinci operandına ilişkin ifadenin değeridir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *foo*()  
{  
 *writeln*(**"foo"**)  
  
 **return** 20  
}  
  
**function** *bar*()  
{  
 *writeln*(**"bar"**)  
 **return** 10  
}  
  
**function** *main*()  
{  
 **let** a = (*foo*(), *bar*())  
  
 *writeln*(a)  
}  
  
  
*main*()

Virgül operatörü operatör öncelik tablosunun en alt seviyesinde olduğundan yukarıdaki işlemde atama işleminin en son yapılması için parantez zorunludur. Virgül operatörünün önce birinci operandının yapılacağı garanti altındadır. Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *foo*()  
{  
 *writeln*("foo")  
  
 return 20  
}  
  
function *bar*()  
{  
 *writeln*("bar")  
 return 10  
}  
  
function *tar*()  
{  
 *writeln*("tar")  
 return 12  
}  
  
function *main*()  
{  
 let a = (*foo*(), *bar*(), *tar*())  
  
 *writeln*(a)  
}  
  
*main*()

**typeof Operatörü**

Bu operatör operandı olan ifadenin türünü string olarak üretir. Parantezli veya parantezsiz kullanılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = 10  
  
 *writeln*(**typeof** a)  
  
 a = **"ankara"** *writeln*(**typeof**(a))  
}  
  
  
*main*()

Diğer operatörler ileride ele alınacaktır.

**Otomatik Tür dönüşümleri**

|  |  |  |  |
| --- | --- | --- | --- |
| **Değer** | **Number** | **String** | **Boolean** |
| false | 0 | "false" | false |
| true | 1 | "true" | true |
| 0 | 0 | "0" | false |
| 1 | 1 | "1" | true |
| "0" | 0 | "0" | true |
| "000" | 0 | "000" | true |
| "1" | 1 | "1" | true |
| NaN | NaN | "NaN" | false |
| Infinity | Infinity | "Infinity" | true |
| -Infinity | -Infinity | "-Infinity" | true |
| "" | **0** | "" | false |
| "20" | 20 | "20" | true |
| "twenty" | NaN | "twenty" | true |
| [ ] | **0** | "" | true |
| [20] | **20** | "20" | true |
| [10,20] | NaN | "10,20" | true |
| ["twenty"] | NaN | "twenty" | true |
| ["ten","twenty"] | NaN | "ten,twenty" | true |
| function(){} | NaN | "function(){}" | true |
| { } | NaN | "[object Object]" | true |
| null | **0** | "null" | false |
| undefined | NaN | "undefined" | false |

**Kontrol Deyimleri**

Akışın kontrolü için kullanılan deyimlere kontrol deyimleri denir.

**if Deyimi**

if deyimi parantez içerisindeki ifadenin true ya da false olmasına göre akışı yönlendiren bir deyimdir. Burada parantez içerisindeki ifade boolean türden olmasa bile otomatik tür dönüşümü kurallarına göre boolean türüne dönüştürülür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = **"ankara"  
  
 if** (a)  
 *writeln*(**"Doğru"**)  
 **else** *writeln*(**"Yanlış"**)  
}  
  
  
*main*()

Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = 10  
  
 **if** (a)  
 *writeln*(**"Doğru"**)  
 **else** *writeln*(**"Yanlış"**)  
}  
  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı ikinci dereceden denklemin katsayılarına göre köklerini bulup ekrana yazdıran findRoots fonksiyonunu yazınız.

***Açıklama:*** *Klavyeden değer okuma NodeJS’ ye özgü olduğundan klavye işlemleri burada ele alınmayacaktır. Karekök alma işlemi için Math.sqrt kullanılabilir*

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *findRoots*(a, b, c)  
{  
 **let** delta = b \* b - 4 \* a \* c  
  
 **if** (delta > 0) {  
 **let** sqrtDelta = ***Math***.sqrt(delta)

**let** x1 = (-b + sqrtDelta) / (2 \* a)  
 **let** x2 = (-b - sqrtDelta) / (2 \* a)  
  
 *writeln*(**`x1=**${x1}**, x2=**${x2}**`**)  
 }  
 **else if** (delta == 0) {  
 **let** x = -b / (2 \* a)  
  
 *writeln*(**`x1=x2=**${x}**`**)  
 }  
 **else** *writeln*(**"Gerçek kök yok"**)  
}  
  
**function** *main*()  
{  
 *findRoots*(1, -3, -18)  
 *findRoots*(1, 1, 1)  
 *findRoots*(1, 4, 4)  
 *findRoots*(1.4, -3.5, -1.8)  
}  
  
  
*main*()

Diğer bir çözüm:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *findRoots*(a, b, c)  
{  
 **let** delta = b \* b - 4 \* a \* c  
  
 **if** (delta >= 0) {  
 **let** sqrtDelta = ***Math***.sqrt(delta)  
 **let** x1 = (-b + sqrtDelta) / (2 \* a)  
 **let** x2 = (-b - sqrtDelta) / (2 \* a)  
  
 *writeln*(**`x1=**${x1}**, x2=**${x2}**`**)  
 }  
 **else** *writeln*(**"Gerçek kök yok"**)  
}  
  
**function** *main*()  
{  
 *findRoots*(1, -3, -18)  
 *findRoots*(1, 1, 1)  
 *findRoots*(1, 4, 4)  
 *findRoots*(1.4, -3.5, -1.8)  
}  
  
*main*()

**Döngü Deyimleri**

ES'de döngü deyimleri genel olarak üçe ayrılır. Ancak ES6 ile birlikte bir döngü deyimi daha eklenmiştir:

1. while döngüleri

2. for döngüsü

3. for-in döngüsü

4. for-of döngüsü

**while Döngüleri**

while döngü deyimleri iki gruba ayrılır:

- Kontrolün başta yapıldığı while döngü deyimi:

- Kontrolün sonda yapıldığı while döngü deyimi:

while döngüsü dendiğinde genelde kontrolün başta yapıldığı while döngü deyimi anlaşılır. Kontrolün sonda yapıldığı while döngü deyimi genel olarak "do-while döngü deyimi" biçiminde söylenir.

**Kontrolün başta yapıldığı while döngü deyimi:**

Bu deyimin genel biçimi şöyledir:

while (<ifade>)

<deyim>

Bu deyimde parantez içerisindeki ifade true ise (boolean türü dışında dönüşüm işleminden sonra) döngü yinelenir. while döngü deyiminde parantez içerisinde ifadenin doğru olup olmadığına akış while döngü deyimine geldiğinde de bakılır. Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}

**function** *main*()  
{  
 **let** i = 0  
 **let** n = 10  
  
 **while** (i < n) {  
 *writeln*(**`i=**${i}**`**)  
 ++i;  
 }  
}  
  
*main*()

while döngü deyimi ile n-kez yinelenen aşağıdaki gibi yazılan bir döngü bazı programcılar tarafından çok sevilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
**function** *main*()  
{  
 **let** n = 4  
  
 **while** (n--) // ~ **while** (n-- > 0) -> Negatif değerler için aynı anlama gelmez  
 *writeln*(**`n=**${n}**`**)   
  
 *writeln*(**`after loop:n=**${n}**`**)  
}  
*main*()  
  
Yukarıdaki döngüde n değerinin döngünün sonunda eski değerinde olmadığına dikkat ediniz.

**Sınıf Çalışması:** Parametresi ile aldığı number türden bir sayının basamak sayısını döndüren digitsCount fonksiyonunu yazınız.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *digitsCount*(val)  
{  
 **if** (!val)  
 **return** 1  
  
 **let** count = 0  
  
 **while** (val) {  
 ++count  
 val = *parseInt*(val / 10)  
 }  
  
 **return** count  
}  
  
**function** *main*()  
{  
 *writeln*(*digitsCount*(123))

*writeln*(*digitsCount*(-123))

*writeln*(*digitsCount*(0))  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı tamsayının tersini döndüren getReverse isimli fonksiyonu yazınız.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *reversed*(val)  
{  
 **let** reverse = 0  
  
 **while** (val) {  
 reverse = reverse \* 10 + val % 10  
 val = *parseInt*(val / 10)  
 }  
   
 **return** reverse  
}  
  
**function** *main*()  
{  
 *writeln*(*reversed*(123))  
 *writeln*(*reversed*(-123))  
}  
  
*main*()

while döngü deyimi ile tipik sonsuz döngü kalıpları:

**while** (**true**) {  
 *//...*}

**while** (1) {  
 *//...*}

**Kontrolün sonda yapıldığı while döngü deyimi (do-while)**

Bu deyimin genel biçimi:

do

<deyim>

while (<ifade>)

Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *digitsCount*(val)  
{  
 **let** count = 0  
  
 **do** {  
 ++count  
 val = *parseInt*(val / 10)  
 } **while** (val !== 0)  
  
 **return** count  
}  
  
**function** *main*()  
{  
 *writeln*(*digitsCount*(0))  
 *writeln*(*digitsCount*(-345))  
}

*main*()

***Anahtar Notlar:*** *Bir sayının basamak sayısı döngü veya yazı kullanmadan aşağıdaki gibi de bulunabilir. Aşağdaki örnekte Math sınıfının abs ve log10 metotları kullanılmıştır. Bu metotların çalışması Java/C# ile aynıdır. Ayrıca kullanılan koşul operatörü de yine Java/C# ile aynıdır:*

*function writeln(a)  
{  
 console.log(a)  
}  
  
function digitsCount(val)  
{  
 return !val ? 1 : parseInt(Math.log10(Math.abs(val))) + 1  
}  
  
function main()  
{  
 writeln(getDigitsCount(0))  
 writeln(getDigitsCount(-345))  
}*

*main()*

**for döngü deyimi**

for döngü deyimi de Java' da olduğu gibi güçlü bir döngü deyimidir. Bu deyimin genel biçimi şöyledir:

for ([1.kısım];[2.kısım]; [3.kısım])

<deyim>

Bu döngüde kısımlar zorunlu değildir. Kısımlar boş bırakılmış olsa bile iki tane noktalı virgül zorunludur.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **for** (**let** i = 0; i < 10; ++i)  
 *writeln*(**`i=**${i}**`**)  
}  
  
*main*()

Kısımların çalışma mantığı Java’dakine çok benzer. 2. kısmın boş bırakılması yine sonsuz döngü anlamına gelmektedir. Ancak okunabilirlik açısından sonsuz döngü kalıbı tüm kısımların boş bırakılması biçiminde yazılır:

**for (;;) {**

**//...**

**}**

for döngü deyimi içerisinde virgül operatörü kullanılarak deyim zenginleştirilebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **for** (**let** i = 0, k = 10; i < 20 && k >= 0; ++i, k -= 2)  
 *writeln*(**`(**${i}**,** ${k}**)`**);  
}  
  
*main*()

***Anahtar Notlar:*** *Virgül ile değişken bildirimi yapıldığında ikinci operand olan bildirim birinci operand olan bildirim ile aynı özelliktedir. Aşağıdaki örnekte i ve k değişkenleri let ile bildirilmiştir. Bu durumda i ve k fonksiyon faaliyet alanındadır:*

*function writeln(a)  
{  
 console.log(a)  
}  
  
function main()  
{  
 let i = 20, k = 10  
  
 writeln(`i=${i}`)  
 writeln(`k=${k}`)  
}  
  
main()  
  
writeln(`k=${k}`) //k is not defined*

*Halbuki virgül kullanılmasaydı k global bildirilmiş olacaktı:*

*function writeln(a)  
{  
 console.log(a)  
}  
  
function main()  
{  
 let i = 20  
 k = 10  
  
 writeln(`i=${i}`)  
 writeln(`k=${k}`)  
}  
  
main()  
  
writeln(`k=${k}`)*

for döngü deyimi zenginleştirilebilir. Güçlü bir deyimdir. Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}

**function** *main*()  
{  
 **for** (**let** i = 0, k = 10; i < 20 && k >= 0; *writeln*(**`(**${i}**,** ${k}**)`**), ++i, k -= 2)  
 ;  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi aldığı bir tamsayının asal olup olmadığını test eden isPrime fonksiyonunu yazınız (Yavaş version)

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *isPrime*(val)  
{  
 **if** (val <= 1)  
 **return false  
  
 let** halfVal = *parseInt*(val / 2)  
  
 **for** (**let** i = 2; i <= halfVal; ++i)  
 **if** (val % i == 0)  
 **return false  
  
 return true**}  
  
**function** *main*()  
{  
 **for** (**let** n = -10; n <= 100; ++n)  
 **if** (*isPrime*(n))  
 *writeln*(n)  
  
 *writeln*(*isPrime*(1000003) ? **"Asal"** : **"Asal değil"**)  
}  
  
*main*()  
  
Yukarıdaki teknik kötüdür. Aşağıdaki teoremle fonksiyon daha etkin yazılabilir.

***Teorem (Eratosten):*** *Bir sayı karekökünden daha küçük olan asal sayıların hiçbirine bölünemiyorsa asaldır. Aksi durumda asal değildir.*

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *isPrime*(val)  
{  
 **if** (val <= 1)  
 **return false  
  
 if** (val % 2 == 0)  
 **return** val == 2  
  
 **if** (val % 3 == 0)  
 **return** val == 3  
  
 **if** (val % 5 == 0)  
 **return** val == 5  
  
 **if** (val % 7 == 0)  
 **return** val == 7  
  
 **for** (**let** i = 11; i \* i <= val; i += 2)  
 **if** (val % i == 0)  
 **return false  
  
 return true**}  
  
**function** *main*()  
{  
 **for** (**let** n = -10; n <= 100; ++n)  
 **if** (*isPrime*(n))  
 *writeln*(n)  
  
 *writeln*(*isPrime*(1000003) ? **"Asal" :** **"Asal değil"**)  
}  
  
*main*()

**break Deyimi**

break deyiminin genel biçimi şu şekildedir:

break [etiket-ismi];

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **for** (**let** i = 20; i < 40; ++i) {  
 **if** (i % 12 == 0)  
 **break**

*writeln*(i);  
 }  
}  
  
*main*()  
  
break deyiminin etiketli biçimi kullanılarak içiçe döngülerden çıkış sağlanabilir. Zaten bunun için tasarlanmıştır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **EXIT\_LOOP**:  
 **for** (**let** i = 20; i < 40; ++i) {  
 **for** (**let** k = 21; k < 50; ++i) {  
 **if** (i % 12 == 0 && k < 25)  
 **break EXIT\_LOOP** *writeln*(**`(**${i}**,** ${k}**)`**);  
 }  
 }  
  
 ***console***.log(**"Tekrar yapıyor musunuz?"**)  
}  
*main*()

Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **EXIT\_LOOP**:  
 **for** (**let** i = 20; i < 40; ++i) {  
 **EXIT\_SECOND\_LOOP**:  
 **for** (**let** j = 20; j < 40; ++j) {  
 **for** (**let** k = 21; k < 50; ++i) {  
 **if** (j > 22)  
 **break EXIT\_SECOND\_LOOP  
  
 if** (i % 12 == 0 && k < 25)  
 **break EXIT\_LOOP** *writeln*(**`(**${i}**,** ${j}**,** ${k}**)`**);  
 }  
 }  
 }  
  
 ***console***.log(**"Tekrar yapıyor musunuz?"**)  
}  
  
*main*()

**continue Deyimi**

continue deyimi döngünün adımını sonlandırmak için kullanılabilir. continue deyiminin genel biçimi:

continue [etiket ismi];

continue deyimi break deyimi kadar kullanılmasa da bazı durumlarda okunabilirliği artırdığı için tercih edilebilir.

Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** sum = 0  
  
 **for** (**let** i = 1; i <= 100; ++i) {  
 **if** (i % 2 == 0)  
 **continue** sum += i  
 }  
  
 ***console***.log(**`Toplam:**${sum}**`**)  
}  
  
*main*()

Etiketli continue deyimi ile bir döngünün adımı içerisindeyken onu kapsayan döngüden devam edilmesi sağlanabilir. Etiketli continue deyimi yapısal programlamada dolayısıyla nesne yönelimli programlamada da tercih edilmemektedir.

**Sınıf Çalışması:** Parametresi ile aldığı n sayısına göre n-inci asal sayıyı döndüren getPrime fonksiyonunu yazınız.

**Çözüm:**

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *isPrime*(val)  
{  
 if (val <= 1)  
 return false  
  
 if (val % 2 == 0)  
 return val == 2  
  
 if (val % 3 == 0)  
 return val == 3  
  
 if (val % 5 == 0)  
 return val == 5  
  
 if (val % 7 == 0)  
 return val == 7  
  
 for (let i = 11; i \* i<= val; i += 2)  
 if (val % i == 0)  
 return false  
  
 return true  
}  
  
function *getPrime*(n)  
{  
 let count = 0  
 let i = 2  
  
 for (; count < n; ++i)  
 if (*isPrime*(i))  
 ++count  
  
 return i - 1  
}  
  
function *main*()  
{  
 *writeln*(*getPrime*(1))  
 *writeln*(*getPrime*(2))  
 *writeln*(*getPrime*(3))  
 *writeln*(*getPrime*(20))  
}  
  
*main*()

**Diğer bir çözüm:**

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *isPrime*(val)  
{  
 if (val <= 1)  
 return false  
  
 if (val % 2 == 0)  
 return val == 2  
  
 if (val % 3 == 0)  
 return val == 3  
  
 if (val % 5 == 0)  
 return val == 5  
  
 if (val % 7 == 0)  
 return val == 7  
  
 for (let i = 11; i \* i<= val; i += 2)  
 if (val % i == 0)  
 return false  
  
 return true  
}  
  
function *getPrime*(n)  
{  
 let count = 0  
 let val = 2  
  
 for (;;) {  
 if (*isPrime*(val))  
 ++count  
  
 if (count == n)  
 return val  
  
 ++val  
 }  
  
}  
  
function *main*()  
{  
 *writeln*(*getPrime*(1))  
 *writeln*(*getPrime*(2))  
 *writeln*(*getPrime*(3))  
 *writeln*(*getPrime*(20))  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı bir sayının Armstrong sayısı olup olmadığını test eden isArmstrong fonksiyonunu yazınız.

**Tanım:** Bir sayının basamaklarının basamak sayıncı kuvvetleri toplamı kendisine eşitse sayı bu sayıya Armstrong sayısı denir:

Örneğin:

153 sayınının basamaklarının küpleri toplamı kendisine eşittir. Bu durumda bu sayı Armstrong sayısıdır.

**Çözüm:**

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *digitsPowSum*(val)  
{  
 let n = *digitsCount*(val)  
 let sum = 0;  
  
 while (val) {  
 sum += *parseInt*(Math.pow(val % 10, n))  
 val = *parseInt*(val / 10)  
 }  
  
 return sum  
}  
  
function *digitsCount*(val)  
{  
 return val ? *parseInt*(Math.log10(Math.abs(val))) + 1 : 1  
}  
  
function *isArmstrong*(val)  
{  
 return val >= 0 && *digitsPowSum*(val) === val  
}  
  
function *main*()  
{  
 for (let n = -10; n <= 999999; ++n)  
 if (*isArmstrong*(n))  
 *write*(`${n} `)  
 *writeln*()  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı bir n değeri için n-inci Fibonacci sayısını döndüren getFibonacciNumber isimli fonksiyonu yazınız.

Fibonacci Sayıları: 0 1 1 2 3 5 8 13 21 …

**Çözüm:**

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *getFibonacciNumber*(n)  
{  
 if (n <= 2)  
 return n - 1  
  
 let prev1 = 1, prev2 = 0, val = 0  
  
 for (let i = 2; i < n; ++i) {  
 val = prev1 + prev2  
 prev2 = prev1  
 prev1 = val  
 }  
  
 return val  
}  
  
function *main*()  
{  
 *writeln*(*getFibonacciNumber*(1))  
 *writeln*(*getFibonacciNumber*(2))  
 *writeln*(*getFibonacciNumber*(3))  
 *writeln*(*getFibonacciNumber*(7))  
 *writeln*(*getFibonacciNumber*(20))  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı bir sayıdan büyük en küçük Fibonacci sayısını döndüren getNextFibonacciNumber fonksiyonunu yazınız.

Örneğin: sayı 12 ise 13 dönecektir, sayı 13 ise 21 dönecektir

**Çözüm:**

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *getNextFibonacciNumber*(val)  
{  
 if (val < 0)  
 return 0  
  
 let prev1 = 1, prev2 = 0, result = 0  
  
 for (;;) {  
 result = prev1 + prev2  
 if (result > val)  
 return result  
 prev2 = prev1  
 prev1 = result  
 }  
}  
  
function *main*()  
{  
 *writeln*(*getNextFibonacciNumber*(-11))  
 *writeln*(*getNextFibonacciNumber*(0))  
 *writeln*(*getNextFibonacciNumber*(1))  
 *writeln*(*getNextFibonacciNumber*(2))  
 *writeln*(*getNextFibonacciNumber*(8))  
 *writeln*(*getNextFibonacciNumber*(9))  
 *writeln*(*getNextFibonacciNumber*(13))  
 *writeln*(*getNextFibonacciNumber*(1000000))  
 *writeln*(*getNextFibonacciNumber*(1346269))  
}  
  
*main*()

**switch Deyimi**

switch deyimi Java'daki klasik switch deyimine çok benzemekle beraber aslında farklılıklar da içermektedir. switch deyiminin Java programlama dilindeki bir takım kısıtları ES'de yoktur.

switch deyiminin aşağı düşme (fall through) özelliği vardır:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *main*()  
{  
 let code = 216  
  
 switch (code) {  
 case 212:  
 *write*("Avrupa ")  
 case 216:  
 *writeln*("İstanbul")  
 break  
 case 312:  
 *writeln*("Ankara")  
 break;  
 case 372:  
 *writeln*("Zonguldak")  
 break  
 default:  
 *writeln*("Geçersiz telefon kodu")  
 }  
  
 *writeln*("Tekrar yapıyor musun?")  
}  
  
*main*()

switch deyimi number türü için yani noktalı sayılar için de kullanılabilir. Ancak eşitlik karşılaştırması yaptığı için dikkatli olunmalıdır:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *main*()  
{  
 let a = 0.1  
 let b = 0.2  
 let val = a + b  
  
 switch (val) {  
 case 0.3:  
 *writeln*("Bir")  
 break  
 case 2.5:  
 *writeln*("İki")  
 break  
 case 3:  
 *writeln*("Üç")  
 break  
 default:  
 *writeln*("Geçersiz değer")  
 }  
  
}  
  
*main*()

***Anahtar Notlar:*** *Gerçek sayılar için doğrudan eşitlik karşılaştırmasında yuvarlama hataları olabileceğinden her zaman doğru sonuç elde edilmez. Eşitlik karşılaştırma işlemi için çeşitli yöntemler kullanılabilir. Örneğin:*

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *main*()  
{  
 let val = 0.1 + 0.2  
 let a = 0.3  
 let delta = 0.00001  
  
 *writeln*(`val=${val}`)  
  
 if (Math.abs(val - a) < delta)  
 *writeln*("Eşit")  
 else  
 *writeln*("Eşit değil")  
}  
  
*main*()

switch deyiminin case bölümlerinin sabit ifadesi olması gerekmez:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
  
function *main*()  
{  
 let val = 1  
 let a = 1  
  
 switch (val) {  
 case a:  
 *writeln*("Bir")  
 break  
 case 2:  
 *writeln*("Iki")  
 break  
 case 3:  
 *writeln*("Uc")  
 break;  
 default:  
 *writeln*("default")  
 }  
  
}  
  
*main*()

Örnek bir menu programı. Program bir html içerisinde çalışabilecek şekilde tasarlanmıştır:

**function** *main*()  
{  
 **EXIT\_LOOP**:  
 **for** (;;) {  
 **var** result = *myprompt*();  
  
 **switch** (result) {  
 **case "1"**:  
 *writeln*(**"Ekle"**);  
 **break**;  
 **case "4"**:  
 **break EXIT\_LOOP**;  
 **default**:  
 *writeln*(**"Hala secmediniz"**);  
 **break**;  
 }  
 }  
  
}

**function** *myprompt*()  
{  
 **var** msg = **"1.Ekle\n"**;  
 msg += **"2.Guncelle\n"**;  
 msg += **"3.Sil\n"**;  
 msg += **"4.Cikis\n"**;  
 msg += **"Secenek:"**;  
  
 **return** *prompt*(msg);  
}  
**function** *writeln*(val)  
{  
 ***document***.write(val + **"</br>"**);  
}  
  
**function** *write*(val)  
{  
 *write*(val);  
}

**Object Türü**

Object, ES' de bir referans türünü temsil eden en temel türdür. Bir object tanımlamak ve yaratmak yaratmak için:

- new operatörü kullanılabilir.

- Küme parantezi kullanılabilir. Buna ilkdeğer verme sentaksı da denebilir. JSON formatı tamamen buna dayanır. Zaten JSON “Javascript Object Notation” demektir.

- ES6 ile birlikte belirli koşullar altında new operatörsüz de nesne yaratılabilir. Ancak, nesnenin buna uygun olarak bildirilmiş olması gerekir. Bu yöntem çok tercih edilmez.

Bir Object referansı ve nokta operatörü ile bir isim yazıldığında bu isim bir property anlamına gelir. Bu isim daha sonra kullanılmak üzere Object’de bildirilmiş olur:

Örneğin:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *main*()  
{  
 let device = new *Object*()  
  
 device.id = 1  
 device.name = "test"  
 device.port = 1234  
 device.ip = "192.168.2.234"  
  
 *displayDevice*(device)  
}  
  
function *displayDevice*(dev)  
{  
 *writeln*(`Device Id:${dev.id}`)  
 *writeln*(`Device name:${dev.name}`)  
 *writeln*(`Device port:${dev.port}`)  
 *writeln*(`Device IP:${dev.ip}`)  
}  
  
*main*()

Object daha basit olarak sadece {} ile de elde edilebilir. Örneğin:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *main*()  
{  
 let device = {}  
  
 device.id = 1  
 device.name = "test"  
 device.port = 1234  
 device.ip = "192.168.2.234"  
  
 *displayDevice*(device)  
}  
  
function *displayDevice*(dev)  
{  
 *writeln*(`Device Id:${dev.id}`)  
 *writeln*(`Device name:${dev.name}`)  
 *writeln*(`Device port:${dev.port}`)  
 *writeln*(`Device IP:${dev.ip}`)  
}  
  
*main*()

Bir Object’in property elemanları küme paranetezi içerisinde de yazılabilir. Bu durumda property elemanlarına değerle küme parantezi içerisinde verilebilir. Burada mutlaka değer verilmesi gerekir. Bu sentaksta değer vermek için = kullanılmaz. ‘:’ atomu kullanılmalıdır.

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}

function *main*()  
{  
 let device = {  
 id: 1,  
 name: "test",  
 port: 1234,  
 ip: "192.168.2.234"  
 }  
  
 *displayDevice*(device)  
  
 device.port = 5765  
  
 *displayDevice*(device)  
}  
  
function *displayDevice*(dev)  
{  
 *writeln*(`Device Id:${dev.id}`)  
 *writeln*(`Device name:${dev.name}`)  
 *writeln*(`Device port:${dev.port}`)  
 *writeln*(`Device IP:${dev.ip}`)  
}  
  
*main*()

Property elamanları tek tırnak veya çift tırnaklar arasında yazılabilir:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}

function *main*()  
{  
 let device = {  
 "id": 1,  
 'name': "test",  
 'port': 1234,  
 'ip': "192.168.2.234"  
  
 }  
  
 *displayDevice*(device)  
}  
  
function *displayDevice*(dev)  
{  
 *writeln*(`Device Id:${dev.id}`)  
 *writeln*(`Device name:${dev.name}`)  
 *writeln*(`Device port:${dev.port}`)  
 *writeln*(`Device IP:${dev.ip}`)  
}  
  
*main*()

Bir object içerisinde fonksiyon tanımlaması yapılabilir:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *main*()  
{  
 let device = {  
 id: 1,  
 name: "test",  
 port: 1234,  
 ip: "192.168.2.234",  
 display: function ()  
 {  
 *writeln*(`Device Id:${this.id}`)  
 *writeln*(`Device name:${this.name}`)  
 *writeln*(`Device port:${this.port}`)  
 *writeln*(`Device IP:${this.ip}`)  
 }  
 }  
  
 device.display()  
 device.port = 5765  
 device.display()  
  
 *displayDevice*(device)  
}  
  
function *displayDevice*(dev)  
{  
 *writeln*(`Device Id:${dev.id}`)  
 *writeln*(`Device name:${dev.name}`)  
 *writeln*(`Device port:${dev.port}`)  
 *writeln*(`Device IP:${dev.ip}`)  
}  
  
*main*()

Burada display property elemanı aslında function türündendir. function türü olduğundan artık fonksiyon çağırma operatörü ile kullanılabilmektedir.

Object'in bir property elemanına global bir fonksiyonun da ismi verilebilir:

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeln*(a)  
{  
 *write*(a === undefined ? '\n' : `${a}\n`)  
}  
  
function *displayDevice*()  
{  
 *writeln*(`Device Id:${this.id}`)  
 *writeln*(`Device name:${this.name}`)  
 *writeln*(`Device port:${this.port}`)  
 *writeln*(`Device IP:${this.ip}`)  
}  
  
  
function *main*()  
{  
 let device = {  
 id: 1,  
 name: "test",  
 port: 1234,  
 ip: "192.168.2.234",  
 display: *displayDevice* }  
  
 device.*display*()  
 device.port = 5765  
 device.*display*()  
  
}  
  
*main*()

XXXXXXXXXXXXXXXXXXXXXXXXXXXX

Burada *displayDevice* fonksiyonu içerisinde this referansı kullanıldığına dikkat ediniz.

Aslında herhangi bir değişken de fonksiyon türünde olabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *display*(val)  
{  
 let isEven = function() {  
 return val % 2 == 0  
 }  
  
 *writeln*(isEven() ? "Çift" : "Tek")  
}  
  
function *main*()  
{  
 *display*(10)  
 *display*(11)  
}  
  
*main*()

Burada isEven değişkeni function türündendir. Yukarıdaki örnekte adeta bir yerel fonksiyon tanımlanmıştır. isEven fonksiyonunun display fonksiyonun parameter değişkenini kullanabildiğine (capture) dikkat ediniz.

Yukarıdaki örnekte isEven fonksiyonu aşağıdaki gibi de tanımlanabilirdi:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *display*(val)  
{  
 function isEven() {  
 return val % 2 == 0  
 }  
  
 *writeln*(isEven() ? "Çift" : "Tek")  
}  
  
function *main*()  
{  
 *display*(10)  
 *display*(11)  
}  
  
*main*()

**Diziler**

ES’ de diziler daha fazla yeteneğe sahip olacak biçimde tasarlanmıştır. Yani ES’ de diziler yalın olarak düşünülmemelidir.

Dizilere ilk değer verilmesi aşağıdaki gibi yapılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
  
**function** *main*()  
{  
 **let** names = [**"ali"**, **"veli"**, **"selami"**];  
  
 *writeln*(names)  
}  
  
  
*main*()

Bir dizi Array türü ile new operatörü kullanılarak yaratılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
  
**function** *main*()  
{  
 **let** names = **new *Array***()  
  
 *writeln*(names)  
}  
  
  
*main*()  
  
Bir diziye ekleme [] operatörü ile yapılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
  
**function** *main*()  
{  
 **let** names = **[]**  
  
 names[0] = **"ali"** names[1] = **"veli"** *writeln*(names)  
}  
  
  
*main*()  
  
Array nesnesi için aralara da yer ayrılabilir. Yani aşağıdaki örnekte değer atanmamış indeksteki elemanlar undefined değerinde olurlar:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
  
**function** *main*()  
{  
 **let** names = **new *Array***()  
  
 names[0] = **"ali"** names[3] = **"veli"** *writeln*(names[1])  
}  
  
  
*main*()

Array nesnesi için başlangıçta yer ayrılabilir. Bu durumda tüm elemanlar undefined değerinde olacaktır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = **new *Array***(10)  
  
 **for** (**let** i = 0; i < 10; ++i)  
 *writeln*(a[i])  
}  
  
  
*main*()

Array nesnesi for döngüsü ile dolaşılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4]  
  
 **for** (**let** i = 0; i < numbers.**length**; ++i)  
 *writeln*(numbers[i])  
}  
  
*main*()

**for-in döngü deyimi**

Bu döngü deyimi ile bir dizi baştan itibaren sonuna kadar dolaşılabilir. Bu döngü deyiminde her adımda index numarası verilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4];  
  
 **for** (**let** i **in** numbers)  
 *writeln*(numbers[i])  
}  
  
*main*()

for-in döngü deyimi ile aslında dizinin diğer elemanları da gezilebilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4];  
  
 numbers.**x** = 90  
 numbers.**name** = **"ali"  
  
 for** (**let** i **in** numbers)  
 *writeln*(`${i}->${ numbers[i]}`)  
}  
  
  
*main*()

Dikkat edilirse for-in döngü deyimi ile elde edilen dizi nesnesinin property elemanları index gibi kullanıldığında property elemanlarının değerleri elde edilmektedir.

**for-of döngü deyimi**

ES6 ile birlikte for-of denilen bir döngü deyimi de eklenmiştir. Bu döngü deyiminde her adımda döngü değişkenine dizinin elemanı atanır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4];  
  
 **for** (**let** val **of** numbers)  
 *writeln*(val)  
}  
  
  
*main*()

for-of döngü deyimi ile yalnızca dizinin elemanları elde edilebilir. Dizi nesnesine ilişkin elemanlar ve değerleri elde edilemez:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let numbers = [1, 2, 3, 4];  
  
 numbers.x = 10  
 numbers.y = "ankara"  
  
 for (let val of numbers)  
 *writeln*(val)  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı bir diziyi tersyüz eden reverse isimli global bir fonksiyon yazınız ve test ediniz. Fonksiyon içerisinde ikinci bir dizi kullanmayınız.

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *swap*(a, i, k)  
{  
 let temp = a[i]  
  
 a[i] = a[k]  
 a[k] = temp  
}  
  
function *myReverse*(a)  
{  
 let halfLen = *parseInt*(a.length / 2)  
  
 for (let i = 0; i < halfLen; ++i)  
 *swap*(a, i, a.length - 1 - i)  
}  
  
function *display*(a)  
{  
 for (let elem of a)  
 *writeln*(elem)  
}  
  
function *main*()  
{  
 let numbers = [1, 2, 3, 4, 5]  
  
 *myReverse*(numbers)  
  
 *display*(numbers)  
}  
  
*main*()

Ya da örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *swap*(a, i, k)  
{  
 let temp = a[i]  
  
 a[i] = a[k]  
 a[k] = temp  
}  
  
function *myReverse*(a)  
{  
 let left = 0  
 let right = a.length - 1  
  
 while (left < right)  
 *swap*(a, left++, right--)  
}  
  
function *display*(a)  
{  
 for (let elem of a)  
 *writeln*(elem)  
}  
  
function *main*()  
{  
 let numbers = [1, 2, 3, 4]  
  
 *myReverse*(numbers)  
  
 *display*(numbers)  
}  
  
*main*()

Ya da örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *display*(a)  
{  
 for (let elem of a)  
 *writeln*(elem)  
}  
  
function *doReverse*(a)  
{  
 a.swap = function (i, k) {  
 let temp = this[i]  
  
 this[i] = this[k]  
 this[k] = temp  
 }  
  
 a.myReverse = function() {  
 let left = 0  
 let right = this.length - 1  
  
 while (left < right)  
 this.swap(left++, right--)  
 }  
 a.myReverse()  
}  
  
function *main*()  
{  
 let numbers = [1, 2, 3, 4]  
  
 *doReverse*(numbers)  
 *display*(numbers)  
}  
  
*main*()

Her fonksiyonun içerisinde argümanların bulunduğu arguments isimli bir dizi referansı kullanılabilir. Böylece değişken sayıda argüman alan fonksiyonlarda kolay (ama dikkatli) olarak yazılabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *writeLine*()  
{  
 for (let arg of arguments)  
 process.stdout.write(`${arg} `)  
  
 process.stdout.write("\n")  
}  
  
function *sum*()  
{  
 let total = 0  
  
 for (let val of arguments)  
 total += val  
  
 return total  
}  
  
function *main*()  
{  
 *writeln*(*sum*(10, 20))  
 *writeln*(*sum*())  
 *writeln*(*sum*(10, 20, 30))  
  
 *writeLine*("///////////////////////////////////")  
 *writeLine*(10, 20, 30)  
 *writeLine*("ankara", "istanbul", "izmir", "zonguldak")  
}  
  
*main*()

***Anahtar Notlar:*** *Yukarıdaki örnekte ekrana yazı basmak için kullanılan* ***console.log*** *ve* ***process.stdout.write*** *fonksiyonları NodeJS platformuna özgüdür. Başka platformlar için başka fonksiyonlar çağrılabilir. Yani ekrana yazan standard ES fonksiyonu yoktur*

**İsimsiz fonksiyonlar ve Lambda ifadeleri**

Bilindiği gibi ES’de fonksiyonlar function anahtar sözcüğü ile bildirilir. function anahtar sözcüğünden sonra isim yazılmazsa bu tarz fonksiyonlara isimsiz fonksiyonlar (anonymous functions) denir.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** *add* = **function** (a, b)  
 {  
 **return** a + b  
 }  
  
 *writeln*(**typeof** *add*)  
 *writeln*(*add*(10, 20))  
}  
  
*main*()

İsimsiz fonksiyonlar bir değişkene atanabilir. Bu durumda bu atanan değişkenin türü function olur.

İsimsiz fonksiyonlar arka planda çağrılacak (callback) fonksiyonlar biçiminde başka bir fonksiyona argüman olarak geçilebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *forEach*(array, fn)  
{  
 **for** (**let** elem **of** array)  
 fn(elem)  
}  
  
**function** *main*()  
{  
 **let** a = [1, 2, 3, -6]  
  
 *forEach*(a, **function** (elem) {  
 *writeln*(**`**${elem \* elem}**`**)  
 })  
  
 *writeln*(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 *forEach*(a, **function** (elem) {  
 *writeln*(elem)  
 })

*writeln("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
 forEach(a, writeln)*  
}  
  
*main*()

Ya da örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *forEach*(array, fn)  
{  
 **for** (**let** elem **of** array)  
 fn(elem)  
}  
  
**function** *main*()  
{  
 **let** a = [1, 2, 3, -6]  
  
 **let** *displaySquare* = **function** (elem) {  
 *writeln*(**`**${elem \* elem}**`**)  
 }  
  
 *forEach*(a, *displaySquare*)  
  
 *writeln*(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
  
 **let** *display* = **function** (elem) {  
 *writeln*(elem)  
 }  
 *forEach*(a, *display*)  
}  
  
*main*()

Ya da örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *transform*(a, b, cb)  
{  
 let length = a.length  
  
 for (let i = 0; i < length; ++i)  
 b[i] = cb(a[i])  
}  
  
function *main*()  
{  
 let a = [1, 2, 3, 4]  
 let b = []  
  
 *transform*(a, b, function (val) {return val \* val})  
  
 for (let val of b)  
 *writeln*(val)  
  
}  
  
*main*()

Fonksiyonlar Object oluşturmak için de kullanılabilir. Yani ES’ de bir Object function anahtar sözcüğü ile de oluşturulabilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *WeatherInfo* = **function** (id, place, country, lat, lng, description) {  
 **this**.**id** = id  
 **this**.**place** = place  
 **this**.**country** = country  
 **this**.**lat** = lat  
 **this**.**lng** = lng  
 **this**.**description** = description  
 **this**.toString = **function** () {  
 **return `**${**this**.**country**}**:**${**this**.**place**}**:**${**this**.**lat**}**;**${**this**.**lng**}**,** ${**this**.**description**}**`** }  
}  
  
**function** *main*()  
{  
 **let** wi = **new** *WeatherInfo*(1, **"Mecidiyeköy"**, **"TR"**, 23.456, 41.234, **"Bulutlu"**)  
  
 *writeln*(wi.toString())  
}  
  
*main*()

Ya da örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *Product* = **function** (name, price, stock)  
{  
 **this**.**name** = name  
 **this**.**price** = price  
 **this**.**stock** = stock  
 **this**.getTotal = **function** () {**return this**.**stock** \* **this**.**price** }  
}   
  
**function** *main*()  
{  
 **let** p1 = **new** *Product*(**"laptop"**, 4000, 34)  
 **let** p2 = **new** *Product*(**"mouse"**, 40, 349)  
  
 *writeln*(**`Total:**${p1.getTotal()}**`**)  
 *writeln*(**`Total:**${p2.getTotal()}**`**)  
}  
  
*main*()

Yukarıdaki kodlarda aslında bir Object için constructor (ctor) yazılmıştır. Bu fonksiyonlar new operatörü ile nesne yaratılırken çağrılır.

***Anahtar Notlar:*** *ES’ de de türler kategori olarak değer türleri ve referans türleri şeklinde iki gruba ayrılır. Referans türleri içerisinde adresler tutulur. Object türleri referans türleridir. Örneğin:*

***function*** *writeln(a)  
{* ***console****.log(a)  
}****let*** *Product =* ***function*** *(name, price, stock)  
{* ***this****.****name*** *= name* ***this****.****price*** *= price* ***this****.****stock*** *= stock* ***this****.getTotal =* ***function*** *() {****return this****.****stock*** *\** ***this****.****price*** *}* ***this****.toString =* ***function*** *() {* ***return `****${****this****.****name****}****:****${****this****.****stock****}* ***\**** *${****this****.****price****}****=****${****this****.****stock*** *\** ***this****.****price****}****`*** *}  
}****function*** *main()  
{* ***let*** *p1 =* ***new*** *Product(****"laptop"****, 4000, 34)* ***let*** *p2 = p1  
  
 writeln(p1.toString())  
 writeln(p2.toString())  
  
 p2.****stock*** *\*= 2  
  
 writeln(p1.toString())  
 writeln(p2.toString())  
}  
  
main()*

***Anahtar Notlar:*** *Aslında referans türleri ES terminolojisinde mutable type olarak geçer. Yani aslında herşeyin adresi tutulur. Yalnızca tutulan adresin türü değişikliğe izin vermiyorsa immutable, veriyorsa mutable type olarak ele alınır. Yani aslında yukarıda ”referans türleri içerisinde adresler tutulur” cümlesi basitleştirmek için yazılmıştır. Temel türler immutable, Object türleri ise mutable type olarak ele alınır.*

Fonksiyonlar kendilerinden önce bildirilen değişkenlerin hepsini yakalayabilirler (capture):

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *foo*(b)  
{  
 **let** a = 10  
  
 **let** *f* = **function** ()  
 {  
 ***console***.log(**`a=**${a}**`**)  
 ***console***.log(**`b=**${b}**`**)  
 }  
  
 *f*()  
}  
  
**function** *main*()  
{  
 *foo*(20)  
  
}  
  
*main*()

Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *displayStatus*(a)  
{  
 let isEven = function () {return a % 2 === 0}  
  
 *writeln*(isEven() ? "Çift" : "Tek")  
}  
  
function *main*()  
{  
 *displayStatus*(10)  
 *displayStatus*(11)  
}  
  
*main*()

Yukarıdaki örnek aşağıdaki gibi de yapılabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *displayStatus*(a)  
{  
 function isEven() {return a % 2 === 0}  
  
 *writeln*(isEven() ? "Çift" : "Tek")  
}  
  
function *main*()  
{  
 *displayStatus*(10)  
 *displayStatus*(11)  
}  
  
*main*()

Fonksiyonlar yakaladıkları değişkenleri değiştirebilirler:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *foo*(b)  
{  
 let a = 10  
  
 let f = function () {  
 *writeln*(`a=${a}`)  
 a \*= 2  
 *writeln*(`b=${b}`)  
 b -= 3  
 }  
  
 f()  
 f()  
}  
  
function *main*()  
{  
 *foo*(20)  
  
}  
  
*main*()

**Lambda İfadeleri**

Lambda ifadeleri fonksiyonel programlama tekniği ile kod yazımı için matematikten programlamaya aktarılmıştır. Lambda ifadeleri anonim fonksiyonların gelişmiş biçimi gibi düşünülebilir. Lambda ifadeleri ile çok karmaşık kodlar yazılabilir. Ancak biz kullanılan genel biçimlerini ele alacağız. Lambda ifadelerinin genel biçimleri:

1. (<değişken listesi>) => ifade

2. (<değişken listesi>) => {..}

3. <değişken> => ifade

4. <değişken> => {…}

5. () => ifade

6. () => {…}

Lambda ifadeleri function türündendir. Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** *f* = (a, b) => a + b

*writeln*(*f*(10, 20))  
}  
  
*main*()

Lambda ifadelerinde kullanılan parametre değişkenleri yalnızca o ifade içerisinde görülebilirdir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** *f* = (a, b) => a + b  
 *writeln*(*f*(10, 20))  
  
 *writeln*(**`a=**${a}**`**) *//error ~ exception*}  
  
*main*()

Lambda ifadeleri kendisinden önce bildirilen değişkenlerin hepsini yakalabilirler:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *foo*(c)  
{  
 **let** *f* = (a, b) => a + b + c

*writeln*(*f*(10, 20))  
}  
  
**function** *main*()  
{  
 *foo*(3)  
}  
  
*main*()

Lambda ifadeleri içerisinde yakalanan değişkenler değiştirilebilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *foo*(c)  
{  
 let f = (a, b) => a + b + c++  
  
 *writeln*(f(10, 20))  
 *writeln*(f(10, 20))  
}  
  
function *main*()  
{  
 *foo*(3)  
}  
  
*main*()

***Anahtar Notlar:*** *ES’de lambda ifadelerine “oklu fonksiyonlar (arrow functions)” da denilmektedir.*

Aşağıdaki örnekte generate fonksiyonu iki tane fonksiyon (callback) almaktadır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *generate*(n, supplier, consumer)  
{  
 while (n--)  
 consumer(supplier())  
}  
  
function *main*()  
{  
 let min = 20  
 let max = 40  
  
 *generate*(10, () => *parseInt*(Math.random() \* (max - min) + min), val => *writeln*(val))  
  
 *writeln*("//////////////////////////////")  
  
 let x = 1  
  
 *generate*(5, () => x++, *writeln*)  
}  
  
*main*()

***Anahtar Notlar:*** *ES’ de genel olarak global olarak yazılan alt programlara fonksiyon, sınıflar (object’ler) içerisinde yazılan alt programlara ise metot denir. Ancak metotlara da fonksiyon denmesi yanlış değildir. Ancak bir fonksiyona metot denmesi yanlış anlaşılabilir.*

**Array Nesnesine Ait Yararlı Metotlar**

Array türüne ait birçok yararlı metot bulunmaktadır. Bu metotların bir kısmı fonksiyonel programlama tekniği ile de uyumlu olarak kullanılabilmektedir. Array türü ile birçok veri yapısına yönelik işlem yapılabilmektedir. Yani Array sınıfına ait fonksiyonlar yardımıyla Array nesnesi birçok veri yapısına uygun olarak çalışabilmektedir.

Array türü istenirse stack (LIFO) gibi kullanılabilir. Array türüne ait push ve pop isimli fonksiyonlar bulunmaktadır.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = []  
  
 a.push(10)  
 a.push(20)  
  
 *writeln*(a.pop())  
 *writeln*(a.pop())  
}  
  
*main*()

Array türünün sort fonksiyonu ile sıralama yapılabilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** a = [-3, 9, -7, 8, 6, 5]  
  
 a.sort()  
 *writeln*(a)  
}  
  
*main*()

Parametresiz sort fonksiyonu değerleri yazıya çevirerek sıralama yapar. Sıralama doğal sıralamadır. Doğal sıralama artan sırada (ascending) sıralamak anlamına gelmektedir. sort sıralamanın nasıl yapılacağına ilişki bir fonksiyon (callback) alabilmektedir. Bu konu ileride ele alınacaktır.

Array nesnenin unshift ve shift metotları ile eleman eklemek ve çıkarmak mümkündür. unshift metodu başa ekleme yapar, shift metodu baştaki elemanı döndürür ve siler:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = **[]**  
  
 **for** (**let** i = 0; i < 10; ++i)  
 numbers.unshift(i)  
  
 *writeln*(numbers)  
 *writeln*(**`Last shifted:**${numbers.shift()}**`**)  
 *writeln*(numbers)  
}  
  
*main*()

Aşağıdaki kodda bir Array unshift ve shift kullanarak boşaltılmış ve her adımda elemanı elde edilmiştir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = **new *Array***()  
  
 **for** (**let** i = 0; i < 10; ++i)  
 numbers.unshift(i)  
  
 **while** (numbers.**length** > 0) {  
 *writeln*(numbers.shift())  
 }  
}  
  
*main*()

Dikkat edilirse unshift ve shift fonksiyonları da elemanın eklendiği yer dışında stack gibi kullanılabilir. Aşağıdaki kodda bir Array push ve shift kullanarak boşaltılmış ve her adımda elemanı elde edilmiştir. Bir FIFO (First In First Out) kuyruk sistemi oluşturulmuştur:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = **new *Array***()  
  
 **for** (**let** i = 0; i < 10; ++i)  
 numbers.push(i)  
  
 **while** (numbers.**length** > 0) {  
 *writeln*(numbers.shift())  
 }  
}  
  
*main*()

Bir Object türünden dizi de yaratılabilir ve for-in ya da for-of döngüsü ile dolaşılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *toString*()  
{  
 **return `**${**this**.**name**}**:**${**this**.**stock** \* **this**.**price**}**`**}  
  
**function** *main*()  
{  
 **let** products = [  
 { **name**: **"laptop"**, **stock**: 6, **price**: 4345, toString: *toString* },  
 { **name**: **"mouse"**, **stock**: 40, **price**: 70, toString: *toString* },  
 { **name**: **"motherboard"**, **stock**: 5, **price**: 500, toString: *toString* },  
 ]  
  
 **for** (**let** index **in** products)  
 *writeln*(products[index].*toString*())  
}  
  
*main*()

for-of döngüsü ile aşağıdaki gibi de yazılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *toString*()  
{  
 **return `**${**this**.**name**}**:**${**this**.**stock** \* **this**.**price**}**`**}  
  
**function** *main*()  
{  
 **let** products = [  
 { **name**: **"laptop"**, **stock**: 6, **price**: 4345, toString: *toString* },  
 { **name**: **"mouse"**, **stock**: 40, **price**: 70, toString: *toString* },  
 { **name**: **"motherboard"**, **stock**: 5, **price**: 500, toString: *toString* },  
 ];  
  
 **for** (**let** product **of** products)  
 *writeln*(product.*toString*());  
}  
  
*main*()

sort metodunun function tür parametreli versiyonu ile sıralama işlemi istenildiği şekilde yapılabilir. sort metodu parametresi ile aldığı fonksiyonun geri dönüş değerine göre artan sırada (ascending) sıralama yapar. Bu fonksiyona karşılaştırma fonksiyonu denir. Fonksiyonun geri dönüş değeri negatif ise birinci parametresi ikinci parametresinden küçük kabul edilir, pozitif ise birinci parametre ikinci parametreden büyük kabul edilir, sıfır ise eşit kabul edilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *toString*()  
{  
 **return `**${**this**.**name**}**:**${**this**.**price**} **\*** ${**this**.**stock**} **=** ${**this**.**stock** \* **this**.**price**}**`**}  
  
**function** *displayProducts*(products)  
{  
 **for** (**let** p **of** products)  
 *writeln*(p.toString())  
}  
  
**function** *main*()  
{  
 **let** products = [  
 { **name**: **"motherboard"**, **stock**: 5, **price**: 500, toString: *toString* },  
 { **name**: **"laptop"**, **stock**: 6, **price**: 4345, toString: *toString* },  
 { **name**: **"mouse"**, **stock**: 40, **price**: 70, toString: *toString* }  
 ]  
  
 *displayProducts*(products)  
 *writeln*(**"\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 products.sort(**function** (p1, p2) {**return** p1.**price** - p2.**price**})  
  
 *displayProducts*(products)  
}  
  
*main*()

Aynı örnek lambda ifadesi kullanılarak da yapılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *toString*()  
{  
 **return `**${**this**.**name**}**:**${**this**.**price**} **\*** ${**this**.**stock**} **=** ${**this**.**stock** \* **this**.**price**}**`**}  
  
**function** *displayProducts*(products)  
{  
 **for** (**let** p **of** products)  
 *writeln*(p.toString())  
}  
  
**function** *main*()  
{  
 **let** products = [  
 { **name**: **"motherboard"**, **stock**: 5, **price**: 500, toString: *toString* },  
 { **name**: **"laptop"**, **stock**: 6, **price**: 4345, toString: *toString* },  
 { **name**: **"mouse"**, **stock**: 40, **price**: 70, toString: *toString* }  
 ]  
  
 *displayProducts*(products)  
 *writeln*(**"\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 products.sort((p1, p2) => p1.**price** - p2.**price**)  
  
 *displayProducts*(products)  
}  
  
*main*()

Aynı yöntemle büyükten küçüğe sıralama karşılaştırma fonksiyonu değiştirilerek yapılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *toString*()  
{  
 **return `**${**this**.**name**}**:**${**this**.**price**} **\*** ${**this**.**stock**} **=** ${**this**.**stock** \* **this**.**price**}**`**}  
  
**function** *displayProducts*(products)  
{  
 **for** (**let** p **of** products)  
 *writeln*(p.toString())  
}  
  
**function** *main*()  
{  
 **let** products = [  
 { **name**: **"motherboard"**, **stock**: 5, **price**: 500, toString: *toString* },  
 { **name**: **"laptop"**, **stock**: 6, **price**: 4345, toString: *toString* },  
 { **name**: **"mouse"**, **stock**: 40, **price**: 70, toString: *toString* }  
 ]  
  
 *displayProducts*(products)  
 *writeln*(**"\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 products.sort((p1, p2) => p2.**price** – p1.**price**)  
  
 *displayProducts*(products)  
}  
  
*main*()

Array nesnesinin slice fonksiyonu ile dizinin belirli bir aralığındaki elemanlar yine Array olarak alınabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [-10, -2, 30, -4, 5]  
 **let** sliced = numbers.slice(2, 3) *//[2, 3)  
  
 writeln*(sliced)  
}  
  
  
*main*()

slice fonksiyonuna negatif index numaraları verilebilir. Bu durumda index numaralarını son elemana göre almaktadır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
 **let** sliced = numbers.slice(-3, -1) *//[2, 4)  
  
 writeln*(sliced)  
}  
  
*main*()

Aslında negatif indeks değerleri için dizinin uzunluğu ile toplandığında elde edilen aralıktaki değerler verilmiş olur.

slice fonksiyonu alt bir dizi elde edilemezse boş bir dizi döndürür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
 **let** sliced = numbers.slice(30, 3)  *writeln*(sliced)  
}  
  
*main*()

Array nesnesinin indexOf fonksiyonu ile aranacak elemanın index numarası elde edilebilir. Eğer bulunamazsa -1 değerine döner:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
 **let** index = numbers.indexOf(2)  
  
 *writeln*(index)  
}  
  
*main*()

indexOf metodunun ikinci parametresi aramaya başlanacak indeks numarası olarak verilebilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let numbers = [1, 2, 3, 4, 5]  
 let index = numbers.indexOf(2, 2)  
  
 *writeln*(index)  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı bir dizi ve dizi içerisindeki eleman için, o elemanın dizide kaç tane olduğunu döndüren getCount isimli fonksiyonu yazınız ve test ediniz. Test için ilk değer verilmiş bir dizi kullanabilirsiniz.

Çözüm:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *getCount*(array, val)  
{  
 **let** count = 0  
 **let** index = -1  
  
 **while** ((index = array.indexOf(val, index + 1)) != -1)  
 ++count  
  
 **return** count  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5, 2, 5, 6, 2, 7]  
 **let** count = *getCount*(numbers, 2)  
  
 *writeln*(**`count=**${count}**`**)  
}  
  
*main*()

Ya da örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *getCount*(array, val)  
{  
 **let** count = 0  
 **let** index = 0  
  
 **while** ((index = array.indexOf(val, index)) != -1) {  
 ++count  
 ++index  
 }  
  
 **return** count  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5, 2, 5, 6, 2, 7]  
 **let** count = *getCount*(numbers, 2)  
  
 *writeln*(**`count=**${count}**`**)  
}  
  
*main*()

Ya da örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *getCount*(array, val)  
{  
 **let** count = 0  
  
 **for** (**let** index = -1; (index = array.indexOf(val, index + 1)) != -1; ++count)  
 ;  
  
 **return** count  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5, 2, 5, 6, 2, 7]  
 **let** count = *getCount*(numbers, 2)  
  
 *writeln*(**`count=**${count}**`**)  
}  
  
*main*()

Ya da örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *getCount*(array, val)  
{  
 **let** count = 0  
  
 **for** (**let** index = 0; (index = array.indexOf(val, index)) != -1; ++count, ++index)  
 ;  
  
 **return** count  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5, 2, 5, 6, 2, 7]  
 **let** count = *getCount*(numbers, 2)  
  
 *writeln*(**`count=**${count}**`**)  
}  
  
*main*()

Fonksiyon indexOf kullanmadan aşağidaki gibi de yazılabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *getCount*(array, val)  
{  
 let count = 0  
  
 for (let elem of array)  
 if (elem === val)  
 ++count  
  
 return count  
}  
  
function *main*()  
{  
 let numbers = [1, 2, 3, 4, 5, 2, 5, 6, 2, 7]  
 let count = *getCount*(numbers, 2)  
  
 *writeln*(`count=${count}`)  
}  
  
*main*()

***Anahtar Notlar:*** *Bir object’in bir fonksiyonu içerisinde o fonksiyonu çağıran referansa erişmek gerekebilir. Bu durumda this anahtar sözcüğü o fonksiyon içerisinde kullanılabilir. Örneğin:*

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *Device* = function (name, host, port) {  
 this.name = name  
 this.host = host  
 this.port = port  
 this.toString = function () {  
 return `[${this.name}] ${this.host}:${this.port}`  
 }  
}  
  
function *main*()  
{  
 let dev = new *Device*("test", "192.168.1.23", 50000)  
  
 *writeln*(dev.toString())  
}  
  
*main*()

*Burada nesne tanımlanırken çağrılan fonksiyon içerisinde (buna ctor denebilir) this referansı kullanılmış böylelikle yaratılacak nesne ilişkin veri elemanları elde edilmiştir. Burada Java/C# bakış açısıyla veri elamanları non-static olarak bildirilmiştir. Nesne yaratmanın bu biçiminde doğrudan static eleman bildirimi yoktur. this referansı bir nesneye ilişkin fonksiyonları zincir olarak çağırmakta da (fluent) kullanılır. Örneğin:*

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *IntVal* = function(val) {  
 this.val = val  
 this.plus = function (val) {  
 this.val += val  
  
 return this  
 }  
  
 this.times = function (val) {  
 this.val \*= val  
  
 return this  
 }  
  
 this.minus = function (val) {  
 this.plus(-val)  
  
 return this  
 }  
}  
  
function *main*()  
{  
 let intVal = new *IntVal*(10)  
  
 intVal.plus(20).times(2).minus(7)  
 *writeln*(intVal.val)  
}  
  
*main*()

***Anahtar Notlar:*** *Geri dönüş değeri mantıksal bir bilgiyi temsil eden (yani doğru ya da yanlış) fonksiyonlara “predicate” terimi kullanılır. Şüphesiz ES’de bu tarz fonksiyonların geri dönüş değerinin Boolean türden olması daha anlamlıdır. Özellikle “callback” alan fonksiyonlar predicate aldıklarını belirtirler. Bu durumda programcı bu fonksiyona verilecek fonksiyonun geri dönüş değeri Boolean olacak şekilde yazar. Callback alan fonksiyon da yaptığı işlemde bu fonksiyonun geri dönüş değerinin Boolean olacağı fikriyle çağırır. Örneğin:*

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *getCount*(predicate)  
{  
 let count = 0  
 let length = arguments.length  
  
 for (let i = 1; i < length; ++i)  
 if (predicate(arguments[i]))  
 ++count  
  
 return count  
}  
  
function *main*()  
{  
 let count = *getCount*(val => val % 2, 30, 20, -2, 5, 9, 8)  
  
 *writeln*(`count=${count}`)  
  
 let n = 4  
  
 count = *getCount*(s => s.length > n, "ankara", "istanbul", "muş", "bolu")  
  
 *writeln*(`count=${count}`)  
}  
  
*main*()

*Burada getCount fonksiyonu “predicate” almaktadır. Kodları inceleyiniz.*

Array nesnesinin find fonksiyonu ile belirli koşula uygun olan ilk eleman bulunabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
 **let** result = numbers.find(val => val % 2 == 0)  
  
 *writeln*(result)  
}  
  
*main*()

find fonksiyonu koşula uygun herhangi bir eleman yoksa undefined değerini döndürür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 23, 3, 4, 5]  
 **let** result = numbers.find(val => val % 2 == 0)  
  
 *writeln*(result == **undefined** ? **"Bulunamadı"**: result)  
}  
  
*main*()

find fonksiyonu aşağıdaki gibi de kullanılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
 **let** result = numbers.find((val, index)=> val % 2 == 0 && index > 2)  
  
 *writeln*(result)  
}  
  
*main*()

Aşağıdaki gibi bir find global fonksiyonu da yazılabilir:

**function** main()

{

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

**let** result = myfind(numbers, (val, index, a) => val %2 == 0 && index > 2);

writeln(result);

}

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *myfind*(a, pred)  
{  
 **for** (**let** i **in** a)  
 **if** (pred(a[i], i, a))  
 **return** a[i]  
  
 **return undefined**}  
  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
 **let** result = *myfind*(numbers, (val, index)=> val %2 == 0 && index > 2)  
  
 *writeln*(result)  
}  
  
*main*()

myFind fonksiyonu nesneye de eklenebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *myFind*(pred)  
{  
 **for** (**let** i = 0; i < **this**.**length**; ++i)

**if** (pred(**this**[i], i, **this**))  
 **return this**[i]  
  
 **return undefined**}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
  
 numbers.myFind = *myFind* **let** result = numbers.myFind((val, index, a) => val %2 == 0 && index > 2)  
  
 *writeln*(result)  
}  
  
*main*()

Array nesnesinin findIndex fonksiyonu ile belirli bir koşula uyan ilk elemanın index numarası elde edilebilir. Eğer koşula uyan eleman bulunamazsa -1 döner:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 2, 3, 4, 5]  
  
 **let** index = numbers.findIndex(val => val % 2 == 0)  
  
 *writeln*(index != -1 ? **`**${index} **numaralı indekste bulundu`** : **"Bulunamadı"**)  
}  
  
*main*()

Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *Person* = **function** (id, name) {  
 **this**.**id** = id  
 **this**.**name** = name  
 **this**.toString = **function** () {  
 **return this**.**name** }  
}  
  
**function** *main*()  
{  
 **let** people = [**new** *Person*(1, **"oğuz"**), **new** *Person*(2, **"muhammed"**), **new** *Person*(3, **"oğuzhan"**)]  
 **let** index = people.findIndex(per => per.**id** == 2)  
  
 *writeln*(index != -1 ? **`**${index} **numaralı indekste** ${people[index].toString()} **bulundu`** : **"Bulunamadı"**)  
}  
  
*main*()

Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *WeatherInfo* = function (place, latitude, longitude, status, degree) {  
 this.place = place  
 this.latitude = latitude  
 this.longitude = longitude  
 this.status = status  
 this.degree = degree  
}  
  
  
function *main*()  
{  
 let weatherInfos = [  
 new *WeatherInfo*("Beykoz", 25.67, 45.78, "Yağmurlu", 20),  
 new *WeatherInfo*("Şile", 25.67, 45.78, "Güneşli", 33),  
 new *WeatherInfo*("Şişli", 25.67, 45.78, "Bulutlu", 18)  
 ]  
  
 let degree = 34  
  
 let index = weatherInfos.findIndex(wi => wi.degree > degree)  
  
 if (index != -1) {  
 let wi = weatherInfos[index]  
 *writeln*(`${wi.place}, ${wi.status}`)  
 }  
 else  
 *writeln*("Koşula uygun hava durumu bulunamadı")  
  
}  
  
*main*()

Array nesnesinin every metodu dizinin tüm elemanları belirli koşula uyarsa true bir tane bile uymayan eleman varsa false değeri ile döner:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [10, 20, 30, 4, 58]  
  
 **if** (numbers.every(val => val % 2 == 0))  
 *writeln*(**"Tüm sayılar çift"**)  
 **else** *writeln*(**"en az bir tane çift olmayan sayı var"**)

}  
  
*main*()

Örneğin:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *SeatInfo* = **function** (number) {  
 **this**.**number** = number  
 **this**.**isEmpty** = **true**}  
  
**function** *main*()  
{  
 **let** seats = [**new** *SeatInfo*(1), **new** *SeatInfo*(2), **new** *SeatInfo*(3)]  
  
 seats.forEach(s => s.**isEmpty** = **false**)  
  
 **if** (seats.every(s => !s.**isEmpty**))  
 *writeln*(**"Uçak dolu"**)  
 **else** *writeln*(**"Boş koltuk var"**)  
}  
  
*main*()

Array nesnesinin some fonksiyonu belirlenen koşula uyan hiç eleman yoksa false değerini döndürür. Koşula uyan en az bir tane eleman varsa true döner:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [1, 21, 30, 41, 5]  
  
 **if** (numbers.some(val => val % 2 == 0))  
 *writeln*(**"En az bir tane çift sayı var"**)  
 **else** *writeln*(**"Hiç çift yok"**)}  
  
*main*()

Örnek:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *SeatInfo* = **function** (number) {  
 **this**.**number** = number  
 **this**.**isEmpty** = **true**}  
  
**function** *main*()  
{  
 **let** seats = [**new** *SeatInfo*(1), **new** *SeatInfo*(2), **new** *SeatInfo*(3)]  
  
 seats.forEach(s => s.**isEmpty** = **false**)  
  
 seats[0].**isEmpty** = **true  
  
 if** (seats.some(s => s.**isEmpty**))  
 *writeln*(**"Boş koltuk var"**)  
 **else** *writeln*(**"Uçak dolu"**)  
}  
  
*main*()

Aşağıdaki örnekte bitmiş ürün olup olmadığı sorgulanmaktadır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *Product* = **function** (name, price, stock)  
{  
 **this**.**name** = name  
 **this**.**price** = price  
 **this**.**stock** = stock  
 **this**.getTotal = **function** () {**return this**.**stock** \* **this**.**price** }  
}  
  
**function** *main*()  
{  
 **let** products = **new *Array***()  
  
 products.push(**new** *Product*(**"laptop"**, 4000, 34))  
 products.push(**new** *Product*(**"mouse"**, 40, 349))  
 products.push(**new** *Product*(**"klavye"**, 30, 0))  
 products.push(**new** *Product*(**"kalem"**, 40, 34))  
  
 *writeln*(products.some(p => p.**stock** <= 0) ? **"Stokta bitmiş ürün var"** : **"Tüm ürünler mevcut"**)  
}  
*main*()

Aşağıdaki örnekte ilk bitmiş ürün aranmaktadır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *Product* = function (name, price, stock)  
{  
 this.name = name  
 this.price = price  
 this.stock = stock  
 this.getTotal = function () {return (this.stock > 0 ? this.stock : 0) \* this.price }  
 this.toString = function (){return `${this.name}:${this.getTotal()}`}  
}  
  
  
function *main*()  
{  
 let products = new Array()  
  
 products.push(new *Product*("laptop", 4000, 34))  
 products.push(new *Product*("mouse", 40, 349))  
 products.push(new *Product*("klavye", 30, -6))  
 products.push(new *Product*("kalem", 40, 34))  
  
 let index = products.findIndex(p => p.stock <= 0)  
  
 if (index != -1)  
 *writeln*(`${products[index].toString()} ürünü stokta yok`)  
 else  
 *writeln*("Tüm ürünler mevcut")  
}  
*main*()

Aşağıdaki örnekte stokta olmayan tüm ürünler başka bir diziye atılarak listelenmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *Product* = **function** (name, price, stock)  
{  
 **this**.**name** = name  
 **this**.**price** = price  
 **this**.**stock** = stock  
 **this**.getTotal = **function** () {**return this**.**stock** \* **this**.**price** }  
}  
  
**function** *main*()  
{  
 **let** products = **new *Array***()  
  
 products.push(**new** *Product*(**"laptop"**, 4000, 0))  
 products.push(**new** *Product*(**"mouse"**, 40, 349))  
 products.push(**new** *Product*(**"klavye"**, 30, 0))  
 products.push(**new** *Product*(**"kalem"**, 40, 34))  
  
 **let** productsNotInStock = **new *Array***(products.**length**)  
  
 **let** index = -1  
   
 **while** (**true**) {  
 index = products.findIndex((p, i) => i >= index + 1 && p.**stock** <= 0)  
  
 **if** (index == -1)  
 **break** productsNotInStock.push(products[index])  
 }  
  
 productsNotInStock.forEach(p => *writeln*(p.**name**))  
}  
*main*()

Array nesnesinin reduce metodu ile dizinin elemanları kullanılarak bir işlem yapılabilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [2, 4, 6, 8]  
 **let** result = numbers.reduce((sum, val) => sum + val)  
  
 *writeln*(result)  
}  
  
*main*()

reduce fonksiyonunun ikinci parametresi biriktirilecek değerin ilk başlayacağı değeri alan parametredir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [2, 4, 6, 8]  
 **let** result = numbers.reduce((sum, val) => sum + val, 6)  
  
 *writeln*(result)  
}  
  
*main*()

Aşağıdaki örnekte dizi içerisindeki yazılar birleştirilmiştir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** numbers = [**"ankara"**, **"istanbul"**, **"izmir"**, **"muğla"**]  
 **let** result = numbers.reduce((sum, val) => sum + val, **"şehirler:"**)  
  
 *writeln*(result)  
}  
  
*main*()

Basit bir myReduce fonksiyonu aşağıdaki gibi yazılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *myReduce*(f, val) {  
 **let** result = 0  
  
 **if** (***arguments***.**length** >= 2)  
 result = val  
  
 **for** (**let** i = 0; i < **this**.**length**; ++i)  
 result = f(result, **this**[i], i, **this**)  
  
 **return** result  
}  
  
**function** *main*()  
{  
 **let** cities = [**"istanbul"**, **"ankara"**, **"izmir"**]  
  
 cities.myReduce = *myReduce* **let** result = cities.myReduce((sum, val) => sum + val, **"şehirler:"**)  
  
 *writeln*(result)  
  
}  
  
*main*()

filter metodu parametresi ile aldığı koşula uyan elemanlardan oluşan diziyi verir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let cities = ["ankara", "istanbul", "izmir", "muş", "bolu"]  
 let result = cities.filter(city => city.length > 4)  
  
 *writeln*(result)  
}  
  
*main*()

Aşağıdaki örnekte stokta bulunmayan ürünlerden oluşan dizi elde edilmiştir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *Product* = function (name, price, stock)  
{  
 this.name = name  
 this.price = price  
 this.stock = stock  
 this.getTotal = function () {return this.stock \* this.price }  
}  
  
function *main*()  
{  
 let products = new Array()  
  
 products.push(new *Product*("laptop", 4000, 0))  
 products.push(new *Product*("mouse", 40, 349))  
 products.push(new *Product*("klavye", 30, 0))  
 products.push(new *Product*("kalem", 40, 34))  
  
 let productsNotInStock = products.filter(p => p.stock <= 0)  
  
 productsNotInStock.forEach(p => *writeln*(p.name))  
 products.forEach(p => *writeln*(p.name))  
}

*main*()

**Sınıf Çalışması:** Parametresi ile aldığı bir dizinin, parametresi aldığı callback predicate fonksiyon için koşula uyan elemanlarının sola koşula uymayanlarını ise saga atan ve ilk koşula uymayan elemanın indeks numarasını döndüren partition isimli global fonksiyonu yazınız.

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *getRandomIntArray*(n, min, max) *//[min, max)*{  
 let a = new Array(n)  
  
 for (let i = 0; i < n; ++i)  
 a[i] = *parseInt*(Math.random() \* (max - min) + min)  
  
 return a  
}  
  
function *gswap*(a, i, k)  
{  
 let temp = a[i]  
  
 a[i] = a[k]  
 a[k] = temp  
}  
  
function *partition*(a, pred)  
{  
 let partitionIndex = 0  
  
 while (partitionIndex < a.length && pred(a[partitionIndex]))  
 ++partitionIndex  
  
 if (partitionIndex === a.length)  
 return partitionIndex  
  
 for (let i = partitionIndex + 1; i < a.length; ++i)  
 if (pred(a[i]))  
 *gswap*(a, i, partitionIndex++)  
  
 return partitionIndex  
}  
  
function *main*()  
{  
 let a = *getRandomIntArray*(10, 1, 100)  
  
 *writeln*(a)  
  
 let partitionPoint = *partition*(a, val => val % 2 === 0)  
  
 *writeln*(a)  
  
 *writeln*(`Partition Point:${partitionPoint}`)  
  
}  
*main*()

partition fonksiyonunda dizinin ilk koşula uymayan elemanına erişim için findIndex fonksiyonu da kullanılabilir:

function *partition*(a, pred)  
{  
 let partitionIndex = 0  
  
 partitionIndex = a.findIndex(val => !pred(val))  
  
 if (partitionIndex === a.length)  
 return partitionIndex  
  
 for (let i = partitionIndex + 1; i < a.length; ++i)  
 if (pred(a[i]))  
 *gswap*(a, i, partitionIndex++)  
  
 return partitionIndex  
}

**Ellipsis Parametreli Fonksiyonlar**

Bilindiği gibi üç tane noktanın yanyana gelmesinden oluşan atoma (…) “ellipsis” denir. ES6 ile birlikte ellipsis fonksiyon parametresi olarak kullanılmaktadır:

function *foo*(...args)  
{  
  
}  
  
function *bar*(a, b, ...args)  
{  
  
}

Ellipsis parametresi fonksiyonun son parametresi olmak zorundadır:

function *tar*(...args, a, b) *//error*{  
  
}

Bu sebeple bir fonksiyonun iki tane ellipsis parametresi olamaz:

function *tar*(...a, ...b) *//error*{  
  
}

Ellipsis parameter bir Array referansı gibi düşünülebilir. Bu durumda aşağıdaki kodda args Array referansı kullanılarak argümanlara erişilmiştir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *foo*(...args)  
{  
 for (let arg of args)  
 *writeln*(arg)  
}  
  
  
function *main*()  
{  
 *foo*(10, 20, 30)  
 *foo*(0, 20);  
  
}  
*main*()

Fonksiyonun ellipsis parametresine argüman geçilemese bile boş dizi geçilmiş olur:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *foo*(a, b, ...args)  
{  
 *writeln*(`a=${a}`)  
 *writeln*(`b=${b}`)  
  
 for (let arg of args)  
 *writeln*(arg)  
}  
  
  
function *main*()  
{  
 *foo*(10, 20)  
  
}  
*main*()

Yani ellipsis parametresi kesinlikle bir dizi parametresidir. Şüphesiz ellipsis parameresine heterojen türler argüman olarak geçirilebilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *foo*(a, b, ...args)  
{  
 *writeln*(`a=${a}`)  
 *writeln*(`b=${b}`)  
  
 for (let arg of args)  
 *writeln*(arg)  
}  
  
  
function *main*()  
{  
 *foo*(10, 20, "ankara", 30, "istanbul")  
  
}  
*main*()

Dikkat edilirse dizi referansı paremetresi ile ellipsis parametresi arasındaki tipik fark şudur: Ellipsis parametresi bir dizi parametresidir, fakat argüman olarak dizi referansı geçilmesi gerekmez. Zaten geçilen argümanlardan dizi yaratılıp ilgili fonksiyona geçilmiş olur. Ellipsis parametresi ile ES6 ile eklenen Lambda ifadelerine ilişkin bir durum da çözülebilmektedir: Lambda ifadelerinin arguments dizisi ya tanımlanmaz ya da her zaman boş bir dizidir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
  
function *main*()  
{  
 let display = () => {  
 for (let elem of arguments)  
 *writeln*(elem)  
 }  
  
 display(10, 20, 30)  
 *writeln*("/////////")  
 display(10)  
}  
  
*main*()

Yukarıdaki problem ellipsis parametresi ile çözülebilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let display = (...args) => {  
 for (let elem of args)  
 *writeln*(elem)  
 }  
  
 display(10, 20, 30)  
 *writeln*("/////////")  
 display(10)  
}  
  
*main*()

***Anahtar Notlar:*** *Lambda ifadelerinin özellikle callback durumunda kullanımında çok fazla deyim kullanılması okunabilirliği/algılanabilirliği bozabilir. Böyle bir durumda callback olarak geçilecek fonksiyon ya dışarıda tanımlanabilir:*

*function writeln(a)  
{  
 console.log(a)  
}*

*let callback = (sum, elem) => {  
 writeln(`sum=${sum}`)  
 writeln(`elem=${elem}`)  
 writeln("///////////")  
  
 return sum + elem  
 }  
  
  
function main()  
{  
 let a = [1, 2, 3, 4, 5, 6]  
   
 let result = a.reduce(callback)  
  
 writeln(`Toplam:${result}`)  
}  
  
main()*

*ya da Lambda ifadesi içerisinde çağrılan bir fonksiyon yazılmalıdır:*

*function writeln(a)  
{  
 console.log(a)  
}  
  
let callback = function (sum, elem, val) {  
 writeln(`sum=${sum}`)  
 writeln(`elem=${elem}`)  
 writeln("///////////")  
  
 return sum + elem + val  
}  
  
function main()  
{  
 let a = [1, 2, 3, 4, 5, 6]  
 let val = 2  
  
 let result = a.reduce((sum, elem) => callback(sum, elem, val))  
  
 writeln(`Toplam:${result}`)  
}  
  
main()*

**Temel Türleri Sarmalayan Nesneler (Sınıflar)**

ES’ de temel türleri sarmalayan (wrapper) nesneler bulunur. Örneğin string türü için *String* sarmalayan nesnesi, number türü için *Number* sarmalayan nesnesi kullanılabilir. Temel türlerin metotları olamayacağından yararlı işlemler için sarmalayan sınıflar düşünülmüştür. Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let a = 10  
  
 *writeln*(typeof a)  
  
 let msg = a.toString(2)  
  
 *writeln*(typeof msg)  
  
 *writeln*(msg)  
}  
  
*main*()

Burada a nın gösterdiği tür için toString fonksiyonu çağrılırken sarmalanır ve *Number* türü olarak işleme girer. Burada a nın gösterdiği tür değişmez. İşlem öncesi otomatik tür dönüşümü ile Number nesnesi olarak sarmalanır.

Temel türlere ilişkin sarmalama new operatörü ile de yapılabilir. Bu durumda artık değer temel tür olmaktan çıkmıştır.

**String Nesnesi**

ES’ de string türünü sarmalayan *String* isimli bir nesne bulunmaktadır. String nesnesi Java’ da olduğu gibi immutable’dır. Yani String nesnesi içerisinde bulunan yazı üzerinde değişiklik yapılamaz. Değişiklik yapan metotlar yeni bir String nesnesi döndürürler. String nesnesinin yararlı fonksiyonları vardır.

String nesnesinin length property elemanı ile içerisinde tutuğu yazının karakter sayısı elde edilebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"ankara"** *writeln*(**`length:**${s.**length**}**`**)  
}  
  
*main*()

length property elemanına değer atanamaz. Atama durumunda error olup olmayacağı garanti değildir. Ancak length property elemanının değişmeyeceği garanti altındadır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let s = "ankara"  
  
 s.length = 70  
  
 *writeln*(s)  
 *writeln*(s.length)  
}  
  
*main*()

***Anahtar Notlar:*** *Array nesnesi immutable bir tür olmadığından length property elemanına değer atanabilir. Örneğin dizinin eleman sayısı büyütülebilir. Bu durumda yeni elemanlar undefined değerinde olurlar:*

*function writeln(a)  
{  
 console.log(a)  
}  
  
function main()  
{  
 let a = [1, 2]  
  
 for (let val of a)  
 writeln(val)  
  
 writeln("//////////////")  
  
 a.length = 7  
  
 for (let val of a)  
 writeln(val)  
}  
  
main()*

*Benzer şekilde length property elemanı ile dizinin boyutu küçültülebilir:*

*function writeln(a)  
{  
 console.log(a)  
}  
  
function main()  
{  
 let a = [1, 2, 4]  
  
 for (let val of a)  
 writeln(val)  
  
 writeln("//////////////")  
  
 a.length = 2  
  
 for (let val of a)  
 writeln(val)  
}  
  
main()*

String sınıfının charAt metodu ile herhangi bir indekteki karakter elde edilebilir.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"ankara"  
  
 for** (**let** i **in** s)  
 *writeln*(s.charAt(i))  
}  
  
*main*()

charCodeAt metodu ile verilen bir index numarasındaki karakterin karakter tablosundaki sıra numarası elde edilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"oğuz"  
  
 for** (**let** i **in** s)  
 *writeln*(s.charCodeAt(i))  
}  
  
*main*()

concat fonksiyonu ile iki string birleştirilebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** name = **"oğuz"  
 let** surname = **" karan"  
  
 let** fullName = name.concat(surname)  
  
 *writeln*(fullName)  
}  
  
*main*()

ES’ de de Java da olduğu gibi + operatörü ile de birleştirme yapılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** name = **"oğuz"  
 let** surname = **" karan"  
  
 let** fullName = name + surname  
  
 *writeln*(fullName)  
}  
  
*main*()

Aşağıdaki örnekte concat metoduna number türü geçirilmiş ve otomatik olarak string türüne dönüşmüştür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** degree = 3.4  
 **let** msg = **"Bugün hava "**.concat(degree).concat(**" derece"**)  
  
 *writeln*(msg)  
}  
  
*main*()

Yukarıdaki kod aşağıdaki gibi de yazılabilirdi:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** degree = 3.4  
 **let** msg = **"Bugün hava "** + degree + **" derece"** *writeln*(msg)  
}  
  
*main*()

indexOf metodu ile bir yazı içerisinde başka bir yazı aranabilir. Bu durumda metot yazı bulunursa ilk bulunan yazının ilk karakterinin index numarası, bulunamazsa -1 değeri döndürür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** text = **"Bugün hava çok güzel. Bu çok güzel havada ders mi yapılır"  
 let** s = **"çok"  
 let** index = text.indexOf(s)  
   
 *writeln*(index != -1 ? **"Aranan yazı "** + index + **" numaralı indekste bulundu"** : **"Bulunamadı"**)  
}  
  
*main*()

indexOf metoduna ikinci argüman olarak aramaya hangi indeksten başlayacağı bilgisi geçilebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** text = **"Bugün hava çok güzel. Bu çok güzel havada ders mi yapılır"  
 let** s = **"çok"  
 let** index = text.indexOf(s, 26)  
   
 *writeln*(index != -1 ? **"Aranan yazı "** + index + **" numaralı indekste bulundu"** : **"Bulunamadı"**)  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı birinci yazının içerisinde parametresi ile aldığı ikinci yazıdan kaç tane olduğunu döndüren countString isimli fonksiyonu yazınız ve test ediniz. Fonksiyon büyük-küçük harf duyarlı olarak yazılacaktır.

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *countString*(s1, s2)  
{  
 let count = 0  
 let index = -1  
  
 for (; (index = s1.indexOf(s2, index + 1)) !== -1; ++count)  
 ;  
  
 return count  
}  
  
function *main*()  
{  
 let text = "Bugün hava çok güzel. Bu çok güzel havada ders mi yapılır"  
 let s = "az"  
 let count = *countString*(text, s)  
  
 *writeln*(count > 0 ? `Aranan yazı ${count} tane var` : "Aranan yazı Bulunamadı")  
}  
  
*main*()

Fonksiyon aşağıdaki gibi de yazılabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *getStringCount* (text, searchText)  
{  
 let count = 0  
  
 for (let index = -1; (index = text.indexOf(searchText, index + 1)) != -1; ++count)  
 ;  
  
 return count  
}  
  
function *main*()  
{  
 let text = "Bugün hava çok güzel. Bu çok güzel havada ders mi yapılır"  
 let s = "çok"  
 let count = *countString*(text, s)  
  
 *writeln*(count)  
}  
  
*main*()

toUpperCase ve toLowerCase fonksiyonları sırasıyla büyük harfe ve küçük harfe dönüştürülmüş yeni String döndürürler:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *countString*(text, searchText, ignoreCase)  
{  
 if (ignoreCase) {  
 text = text.toLowerCase()  
 searchText = searchText.toLowerCase()  
 }  
  
 let count = 0  
  
 for (let index = -1; (index = text.indexOf(searchText, index + 1)) !== -1; ++count)  
 ;  
  
 return count  
}  
  
function *main*()  
{  
 let text = "Bugün hava çok güzel. BuÇok güzel havada ders mi yapılır"  
 let s = "çok"  
 let count = *countString*(text, s, true)  
  
 *writeln*(count)  
}  
  
*main*()

startsWith ve endsWith metotları sırasıyla bir String’in bir yazı ile başlayıp başlamadığını veya bitip bitmediğini test etmek için kullanılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *concatIfNotStarts*(s1, s2)  
{  
 **return** s1.startsWith(s2) ? s1 : s2 + s1  
}  
  
**function** *concatIfNotEnds*(s1, s2)  
{  
 **return** s1.endsWith(s2) ? s1 : s1 + s2  
}  
  
**function** *main*()  
{  
 **let** url = **"www.csystem.org"  
 let** email = **"oguzkaran@csystem.org"** url = *concatIfNotStarts*(url, **"http://"**)  
 email = *concatIfNotEnds*(email, **".org"**)  
  
 *writeln*(url)  
 *writeln*(email)  
}  
  
*main*()

fromCharCode metodu karakter sıra numarasından karakteri bir String olarak elde etmek için kullanılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
  
**function** *main*()  
{  
 **let** code = 67  
  
 *writeln*(***String***.fromCharCode(code))  
}  
  
*main*()

includes metodu bir yazı içerisinde parametresi aldığı yazının bulunup bulunmadığını test etmek için kullanılabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let str = "Bugün hava çok güzel"  
 let s = "az"  
  
  
 *writeln*(str.includes(s) ? "Yazı var" : "Yazı yok")  
}  
  
*main*()

repeat metodu bir yazıdan istenildiği miktarda birleştirilmiş yeni bir yazı elde etmek için kullanılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
  
**function** *main*()  
{  
 **let** s = **"ali"  
  
 let** str = s.repeat(10)  
  
 *writeln*(str)  
}  
  
*main*()

slice, substr ve substring fonksiyonları bir yazı içerisinden bir parçasını yazı olarak elde etmekte kullanılır. substring fonksiyonu belirli bir indeksten itibaren ikinci parametresi ile aldığı indekse kadar yazıyı döndürür.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"ankara"  
  
 let** str = s.substring(2, 5) *//[2, 5)  
  
 writeln*(str)  
}  
  
*main*()

substr fonksiyonun ikinci parametresi birinci parametresi ile verilen indeksten itibaren kaç tane karakterin alınacağını belirtir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"ankara"  
 let** str = s.substr(2, 3)  
  
 *writeln*(str)  
}  
  
*main*()

substr ve substring fonksiyonlarının index değerleri negative de olabilir. Bu durum bazı interpreter implmentasyonlarında error’ a yol açabilir.

slice fonksiyonu substr fonksiyonu gibidir. Ancak negatif indeks değerleri için de kullanılabilir. Negatif indeks için yazının karakter uzunluğu ile negatif indeks değeri toplanarak elde edilen indeks değerinden itibaren yazı elde edilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"ankara"  
 let** str = s.slice(-4) *//length - 4  
  
 writeln*(str)  
}  
  
*main*()

split metodu verilen ayraç veya ayraçlara gore yazıyı parçalar ve yazının parçalarından oluşan diziyi döndürür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"Bugün hava çok güzel"  
 let** words = s.split(**" "**)  
  
 **for** (**let** word **of** words)  
 *writeln*(word)  
}  
  
*main*()

split metodu düzenli ifade (regular expression) alarak da çalışabilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"Bugün.hava.çok, güzel"  
 let** words = s.split(/[ .]/)  
  
 **for** (**let** word **of** words)  
 *writeln*(word)  
}  
  
*main*()

Düzenli ifade / ve / arasında verilmelidir. Birden fazla karakter ayraç olarak kullanılacaksa [] arasında yazılmalıdır. Yukarıdaki örnekte ayraçlar yanyana yazıldığında yanyana yazılan ayraçlar arasındaki boş string de diziye eklenir. Boş string’in eklenmemesi için ]’den sonra + konmalıdır. Bu işlem düzenli ifadeler için “buradaki karakterlerden bir veya birden fazlası” anlamına gelmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"Bugün.hava.çok, güzel"  
 let** words = s.split(/[ .]/)  
  
 **for** (**let** word **of** words)  
 *writeln*(word)  
}  
  
*main*()

[ veya ] ayraç olarak kullanılacaksa ters bölü ile yazılmalıdır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** s = **"Bugün]hava.çokgüzel"  
 let** words = s.split(/[ .\]]+/)  
  
 **for** (**let** word **of** words)  
 *writeln*(word)  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı bir dizinin elemanlarını aralarında ikinci parametresi ile aldığı başka bir string olacak şekilde birleştirilmiş string ile geri dönden join isimli fonksiyonu yazınız ve test ediniz.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *join*(array, delim)  
{  
 **let** s = array.reduce((result, s) => result + s + delim, **""**)  
  
 **return** s.substr(0, s.**length** - delim.**length**)  
}

**function** *main*()  
{  
 **let** names = [**"oğuz"**, **"ali"**, **"kaan"**, **"necati"**]  
  
 *writeln*(*join*(names, **":"**))  
}  
  
*main*()

Array sınıfının join metodu ile yukarıdaki fonksiyon yazılmadan da birleştirme işlemi yapılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** names = [**"oğuz"**, **"ali"**, **"kaan"**, **"necati"**]  
  
 *writeln*(names.join(**":"**))  
}  
  
*main*()

join metoduna argüman geçilmezse dizinin elemanları aralarına virgül koyarak birleştirip yazıyı döndürür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** names = [**"oğuz"**, **"ali"**, **"kaan"**, **"necati"**]  
  
 *writeln*(names.join())  
}  
  
*main*()

Ayraç koymadan birleştirmek için join metoduna boş string (“”) argüman olarak geçilmelidir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** names = [**"oğuz"**, **"ali"**, **"kaan"**, **"necati"**]  
  
 *writeln*(names.join(**""**))  
}  
  
*main*()

ES6 ile birlikte String sabitine aşağıdaki gibi raw işlemi yapıldığında yazının içerisindeki escape sequence karakterlerin kendi anlamında kullanılması engellenmiş olur:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 *writeln*(String.raw`c:\test\names.txt`)  
}  
  
*main*()

raw işleminde yazının `backtick` karakteri ile verilmesi gerekir.

**Date nesnesi (sınıfı)**

ES’ de tarih zaman işlemleri için Date isimli bir nesne bulunmaktadır. Date nesnesi yaratmak için ctor çeşitli argümanlar ile kullanılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** now = **new *Date***()  
  
  
 *writeln*(now.toString())  
}  
  
*main*()

Date nesnesi parametresiz olarak yaratılırsa çalışılan sistemin o anki tarih-zaman bilgisi elde edilir. toString metodu tarih-zamana ilişkin standart yazı bilgisini döndürür. Bu zaman sistemin zamanıdır.

Date nesnesinin ctor’u yıl, ay, gün, saat, dakika, saniye, milisaniye değerlerini parametre olarak alabilir. Zamana ilişkin parametrelerin default değerleri sıfırdır, tarihe ilişkin parametrelerin default değerleri ay için 1(bir) ve gün için de 1(bir) değeridir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** now = **new *Date***(2019, 2, 27, 19, 26, 34, 234)  
  
 *writeln*(now.toString())  
}  
  
*main*()

***Anahtar Notlar:*** *Date nesnesi ay bilgisini [0, 11] aralığında alır. Buna dikkat edilmelidir.*

Date sınıfının ctor’una milisaniye değeri de parameter olarak geçilebilir.

Date nesnesinin getDay fonksiyonu tarihin haftanın hangi gününe geldiği bilgisini verir.

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** now = **new *Date***(2019, 1, 24)  
  
 *writeln*(now.getDay())  
}  
  
*main*()

getDay fonksiyonu haftanın günlerini [0, 6] aralığında sıfır pazar ve 6 cumartesi olacak şekilde değer verir. Buradaki isim kötü tasarlanmıştır. Dikkat edilmelidir.

getDate fonskiyonu ile o aya ilişkin gün bilgisi elde edilebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** now = **new *Date***(2019, 1, 24)  
  
 *writeln*(now.getDate())  
}  
  
*main*()

getYear fonksiyonu deprecated olmuştur. Bu fonksiyonu yerine getFullYear fonksiyonu kullanılmalıdır:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** now = **new *Date***(2019, 3, 30)  
  
 *writeln*(now.getFullYear())  
}  
  
*main*()

***Anahtar Notlar:*** *Programlamada “deprecated” terimi artık programlamada tercih edilmemesi gereken durumlar için kullanılmaktadır. Deprecated olmuş bir araç (metot, sınıfı vs.) durumuna göre çalışmayabilir. Genel kural deprecated olmuş aracın kullanılmamasıdır. Bazı ES yorumlayıcıları deprecated fonksiyonları silebilir. Bu durumda kod taşınabilir olmaz.*

getTime fonksiyonu 1.1.1970 gece yarısı ile o tarih arasındaki milisaniye değerini döndürür:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **let** now = **new *Date***()  
  
 *writeln*(now.getTime())  
}  
  
*main*()

Tüm get fonksiyonların set fonksiyonu karşılığı da bulunmaktadır.

**Sınıf Çalışması:** 17.08.1999 senesinde 03:02:00 zamanında meydana gelen deprem ile şu anki zaman arasında tam olarak kaç yıl geçtiğini hesaplayan programı yazınız.

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *runEQApplication*()  
{  
 let eqDate = new Date(1999, 7, 17, 3, 2)  
 let now = new Date()  
 let years = (now.getTime() - eqDate.getTime()) / (1000 \* 60 \* 60 \* 24 \* 365)  
  
 *writeln*(`Deprem olalı ${years} yıl geçti`)  
}  
  
function *main*()  
{  
 *runEQApplication*()  
}  
  
*main*()

Date nesnesinin now fonksiyonu 1.1.1970 ile çağrıldığı zaman arasında milisaniye değerini döndürür. Yukarıdaki örnek now fonksiyonu ile de yapılabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *runEQApplication*()  
{  
 let eqDate = new Date(1999, 7, 17, 3, 2)  
 let years = (Date.now() - eqDate.getTime()) / (1000 \* 60 \* 60 \* 24 \* 365)  
  
 *writeln*(`Deprem olalı ${years} yıl geçti`)  
}  
  
function *main*()  
{  
 *runEQApplication*()  
}  
  
*main*()

**Sınıf Çalışması:** Aşağıdaki elemanları içeren bir Person sınıfını fonksiyon ile oluşturunuz.

* Ad soyad
* T.C. Kimlik No
* Doğum tarihi
* Adres

Person nesnelerinden oluşan bir dizi yaratınız. Bu diziyi parametre olarak alıp kişilerin yaş ortalamasını bulan averageAges isimli global fonksiyonu yazınız.

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *Person* = function (name, citizenId, birthDate, address) {  
 const DIVISOR = 1000 \* 60 \* 60 \* 24 \* 365  
 this.name = name  
 this.citizenId = citizenId  
 this.birthDate = birthDate  
 this.address = address  
 this.getAge = function () {  
 return (Date.now() - this.birthDate.getTime()) / DIVISOR  
 }  
}  
  
function *displayPeople*(people)  
{  
 for (let person of people)  
 *writeln*(`${person.name}, ${person.getAge()}`)  
}  
  
function *averageAges*(people)  
{  
 let sum = 0  
  
 for (let person of people)  
 sum += person.getAge()  
  
 return sum / people.length  
}  
  
function *run*()  
{  
 let people = [  
 new *Person*("Ali", "123456789122", new Date(1989, 5, 23), "istanbul"),  
 new *Person*("Oğuz", "123456789126", new Date(1976, 8, 10), "zonguldak"),  
 new *Person*("Veli", "123456789125", new Date(2000, 8, 9), "izmir")  
 ]  
  
 *displayPeople*(people)  
 *writeln*(`Yaş ortalaması:${*averageAges*(people)}`)  
  
}  
  
function *main*()  
{  
 *run*()  
}  
  
*main*()

Date nesnesinin bir grup UTC (Universal Time Clock) zamanına yönelik bilgileri döndüren getUTCXXXX metotları vardır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let now = new Date()  
  
 *writeln*(`${now.getUTCDate()}.${now.getUTCMonth() + 1}.${now.getUTCFullYear()} ${now.getUTCHours()}:${now.getUTCMinutes()}:${now.getUTCSeconds()}`)  
 *writeln*(`${now.getDate()}.${now.getMonth() + 1}.${now.getFullYear()} ${now.getHours()}:${now.getMinutes()}:${now.getSeconds()}`)  
}  
  
*main*()

Date nesnesi aldığı tarih zamana ilişkin değerlere yönelik gerçerlilik kontrolü yapmaz. Verilen değerleri uygun tarihleri elde edecek işlemi yapar:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let date = new Date(2020, 34, 234)  
  
 *writeln*(`${date.toDateString()}`)  
}  
  
*main*()

Bu durumda programcı tarih geçerliliği için kendisi bir fonksiyon yazmalıdır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let daysOfMonths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]  
  
let *isLeapYear* = year => year % 4 === 0 && year % 100 !== 0 || year % 400 === 0  
let *isValidForBounds* = (val, min, max) => min <= val && val <= max  
let *isValidHour* = val => *isValidForBounds*(val, 0, 23)  
let *isValidMinute* = val => *isValidForBounds*(val, 0, 59)  
let *isValidSecond* = val => *isValidForBounds*(val, 0, 59)  
let *isValidTime* = (hour, minute, second) => *isValidHour*(hour) && *isValidMinute*(minute) && *isValidSecond*(second)  
  
let *isValidDate* = function (day, month, year) {  
 if (day < 1 || day > 31 || month < 0 || month > 11)  
 return false  
  
 let days = month === 1 && *isLeapYear*(year) ? 29 : daysOfMonths[month]  
  
 return day <= days  
}  
  
function *main*()  
{  
 let day = 29  
 let month = 1  
 let year = 2019  
 let hour = 22  
 let minute = 7  
 let second = 56  
  
 if (*isValidDate*(day, month, year) && *isValidTime*(hour, minute, second)) {  
 let date = new Date(year, month, day, hour, minute, second)  
  
 *writeln*(date.toString())  
 }  
 else  
 *writeln*("Geçersiz tarih/zaman bilgileri")  
}  
  
*main*()

Şüphesiz kullanılan ortamda buna yönelik yardımcı metotlar bulunabilir.

Date nesnesi immutable değildir. setXXX metotları tarih/zamana ilişkin değerlerin değiştirilmesi için kullanılır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let now = new Date()  
  
 *writeln*(now.toString())  
  
 now.setDate(78) *// Dikkat geçerlilik kontrolü yapılmıyor  
  
 writeln*(now.toString())  
}  
  
*main*()

Bu metotlar da geçerlilik kontrolü yapmazlar

Date nesnesinin (sınıfının) diğer fonksiyonları dökümanlardan incelenebilir.

**Rasgele Sayı Üretimi**

ES’ de rasgele sayı üretimi için bir nesne bulunmamaktadır. Rastgele sayı üretimi Math nesnesinin random isimli fonksiyonu ile yapılabilir. Random fonksiyonu [0, 1) aralığında rasgele belirlenmiş bir sayı üretir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *main*()  
{  
 **for** (**let** i = 0; i < 10; ++i)  
 *writeln*(***Math***.random())  
}  
  
*main*()

Belirli bir aralıkta rasgele sayılar programcı tarafından yazılan fonksiyonlar ile elde edilebilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *randomBoolean*()  
{  
 **return** *randomInt*(0, 2) == 1}  
  
**function** *randomInt*(min, max) *//[min, max)*{  
 **return** *parseInt*(*randomNumber*(min, max))  
}  
  
**function** *randomNumber*(min, max) *//[min, max)*{  
 **return *Math***.random() \* (max - min) + min  
}  
  
**function** *main*()  
{  
 **let** min = 10, max = 20  
  
 **for** (**let** i = 0; i < 10; ++i)  
 *writeln*(*randomInt*(min, max))  
  
 *writeln*(**"\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 **for** (**let** i = 0; i < 10; ++i)  
 *writeln*(*randomNumber*(min, max))  
  
 *writeln*(**"\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 **for** (**let** i = 0; i < 10; ++i)  
 *writeln*(*randomBoolean*())  
}  
  
*main*()

Bu fonksiyonlar Lambda ifadeleri kullanarak da yazılabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *randomNumber* = (min, max) => Math.random() \* (max - min) + min *//[min, max)*let *randomInt* = (min, max) => *parseInt*(*randomNumber*(min, max)) *// [min, max)*let *randomBoolean* = () => *randomInt*(0, 2) === 1  
  
function *main*()  
{  
 let min = 10, max = 20  
  
 for (let i = 0; i < 10; ++i)  
 *writeln*(*randomInt*(min, max))  
  
 *writeln*("\*\*\*\*\*\*\*\*\*\*\*\*")  
 for (let i = 0; i < 10; ++i)  
 *writeln*(*randomNumber*(min, max))  
  
 *writeln*("\*\*\*\*\*\*\*\*\*\*\*\*")  
 for (let i = 0; i < 10; ++i)  
 *writeln*(*randomBoolean*())  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı iki tane yıl arasında rasgele tarih döndüre randomDate isimli fonskiyonu yazınız.

Açıklamalar:

* Fonksiyon tek argüman aldığında sadece o argümana ilişkin yıl içerisinde rasgele tarih üretecektir.
* Fonksiyon argümansız çağrıldığında systemin zamanına ilişkin yıl içerisinde rasgele tarih üretecektir
* Açıklananlar dışından fonksiyon içerisinde başka bir control yapılamayacaktır
* Test kodu için bilgiyi ekrana İngilizce ve Türkçe olarak aşağıdaki formatlarda yazdıran displayDateTR ve displayDateEN isimli fonksiyonları yazınız:

İngilizce format: Jul 11th 1983 Mon

Türkçe format: 11 Temmuz 1983 Pazartesi

function *writeln*(a)  
{  
 console.log(a)  
}  
  
let *randomNumber* = (min, max) => Math.random() \* (max - min) + min *//[min, max)*let *randomInt* = (min, max) => *parseInt*(*randomNumber*(min, max)) *// [min, max)*let *randomBoolean* = () => *randomInt*(0, 2) === 1  
  
let daysOfMonths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]  
let monthsTR = ["Ocak", "Şubat", "Mart", "Nisan", "Mayıs", "Haziran", "Temmuz", "Ağustos", "Eylül", "Ekim", "Kasım", "Aralık"]  
let daysOfWeekTR = ["Pazar", "Pazartesi", "Salı", "Çarşamba", "Perşembe", "Cuma", "Cumartesi"]  
let monthsEN = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]  
let daysOfWeekEN = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]  
  
let *isLeapYear* = year => year % 4 === 0 && year % 100 !== 0 || year % 400 === 0  
  
function *getRandomDate*(minYear, maxYear)  
{  
 let year = *randomInt*(minYear, maxYear + 1)  
 let month = *randomInt*(0, 12)  
 let days = month === 1 && *isLeapYear*(year) ? 29 : daysOfMonths[month]  
 let day = *randomInt*(1, days + 1)  
  
 return new Date(year, month, day)  
}  
  
function *randomDate*(minYear, maxYear) *//[minYear, maxYear]*{  
 let min = minYear  
 let max = min  
  
 if (arguments.length >= 2) {  
 min = minYear  
 max = maxYear  
 }  
 else if (arguments.length === 0)  
 min = max = new Date().getFullYear()  
  
 return *getRandomDate*(min, max)  
}  
  
function *getPostFix*(day)  
{  
 let postFix = "th"  
  
 switch (day) {  
 case 1:  
 case 21:  
 case 31:  
 postFix = "st"  
 break  
 case 2:  
 case 22:  
 postFix = "nd"  
 break  
 case 3:  
 case 23:  
 postFix = "rd"  
 }  
  
 return postFix  
}  
  
function *displayDateTR*(date)  
{  
 *writeln*(`${date.getDate()} ${monthsTR[date.getMonth()]} ${date.getFullYear()} ${daysOfWeekTR[date.getDay()]}`)  
}  
  
function *displayDateEN*(date)  
{  
 *writeln*(`${monthsEN[date.getMonth()]} ${date.getDate()}${*getPostFix*(date.getDate())} ${date.getFullYear()} ${daysOfWeekEN[date.getDay()]}`)  
}  
  
function *main*()  
{  
 for (let i = 0; i < 5; ++i) {  
 let date = *randomDate*(2020)  
  
 *displayDateTR*(date)  
 *displayDateEN*(date)  
 *writeln*("/////////////////")  
 }  
}  
  
*main*()

**Sınıf Çalışması:** Bir paranın yazı gelme olasılığını yaklaşık olarak hesaplayan program yazınız.

**Çözüm-1:**

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *randomBoolean*()  
{  
 return *randomInt*(0, 2) === 1  
}  
  
function *randomInt*(min, max) *//[min, max)*{  
 return *parseInt*(*randomNumber*(min, max))  
}  
  
function *randomNumber*(min, max) *//[min, max)*{  
 return Math.random() \* (max - min) + min  
}  
  
function *getCoinTailProbability*(n)  
{  
 let count = 0  
  
 for (let i = 0; i < n; ++i)  
 count += *randomInt*(0, 2)  
  
 return count / n  
}  
  
function *main*()  
{  
 let n = 100000  
  
 *writeln*(`p=${*getCoinTailProbability*(n)}`)  
}  
  
*main*()

**Çözüm-2:**

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *randomBoolean*()  
{  
 return *randomInt*(0, 2) === 1  
}  
  
function *randomInt*(min, max) *//[min, max)*{  
 return *parseInt*(*randomNumber*(min, max))  
}  
  
function *randomNumber*(min, max) *//[min, max)*{  
 return Math.random() \* (max - min) + min  
}  
  
function *getCoinTailProbability*(n)  
{  
 let count = 0  
  
 for (let i = 0; i < n; ++i)  
 if (*randomBoolean*())  
 ++count  
  
 return count / n  
}  
  
function *main*()  
{  
 let n = 100000  
  
 *writeln*(`p=${*getCoinTailProbability*(n)}`)  
}  
  
*main*()

**Sınıf Çalışması:** *Craps* hemen hemen dünyanın her yerinde bilinen, iki zarla oynanan bir oyundur.

Oyunun kuralları şu şekildedir :

Zarları atacak oyuncu oyunu kasaya karşı oynar. Atılan iki zarın toplam değeri

1. ***7*** ya da ***11*** ise oyuncu kazanır.
2. ***2***, ***3***, ***12*** ise oyuncu kaybeder. (Buna ***craps*** denir!)
3. İki zarın toplam değeri yukarıdakilerin dışında bir değer ise (yani ***4***, ***5***, ***6***, ***8***, ***9***, ***10***) oyun şu şekilde devam eder:

Oyuncu aynı sonucu buluncaya kadar zarları tekrar atar. Eğer aynı sonucu bulamadan önce oyuncu ***7*** atarsa (yani atılan iki zarın toplam değeri ***7*** olursa) oyuncu kaybeder.

Eğer ***7*** gelmeden önce oyuncu aynı sonucu tekrar atmayı başarırsa , kazanır.

Birkaç örnek :

Oyuncu zarları attı, zarların toplam değeri :

***11*** oyuncu kazandı. Yeni oyun oynanacak.

***3*** oyuncu kaybetti. Yeni oyun oynanacak.

***12*** oyuncu kaybetti. Yeni oyun oynanacak.

***7*** oyuncu kazandı. Yeni oyun oynanacak.

***9*** sonuç belli değil, oyuncu tekrar zar atacak.

***8*** sonuç belli değil, oyuncu tekrar zar atacak.

***11*** sonuç belli değil oyuncu tekrar zar atacak.

***5*** sonuç belli değil oyuncu tekrar zar atacak.

***9*** oyuncu kazandı. (***7*** atmadan aynı zarı tekrar attı). Yeni oyun oynanacak.

***6*** sonuç belli değil, oyuncu tekrar zar atacak.

***3*** sonuç belli değil, oyuncu tekrar zar atacak.

***10*** sonuç belli değil, oyuncu tekrar zar atacak.

***7*** oyuncu kaybetti. (Aynı zarı tekrar atamadan ***7*** geldi)

Bu oyunu simüle ederek bilgisayara örneğin ***100.000*** kez oynatan, ve oyuncunun kazanma olasılığını hesaplayan programı yazınız.

**Çözüm:**

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *randomBoolean*()  
{  
 return *randomInt*(0, 2) === 1  
}  
  
function *randomInt*(min, max) *//[min, max)*{  
 return *parseInt*(*randomNumber*(min, max))  
}  
  
function *randomNumber*(min, max) *//[min, max)*{  
 return Math.random() \* (max - min) + min  
}  
  
let *getDiceSum* = () => *randomInt*(1, 7) + *randomInt*(1, 7)  
  
function *getStatusByValue*(val)  
{  
 for (;;) {  
 let diceSum = *getDiceSum*()  
  
 if (diceSum === 7)  
 return false  
  
 if (diceSum === val)  
 return true  
 }  
}  
  
function *playCraps*()  
{  
 let sum = *getDiceSum*()  
  
 switch (sum) {  
 case 7:  
 case 11:  
 return true  
 case 2:  
 case 3:  
 case 12:  
 return false  
 default:  
 return *getStatusByValue*(sum)  
 }  
}  
  
function *runSimulation*(n)  
{  
 let winCount = 0  
  
 for (let i = 0; i < n;++i) {  
 if (*playCraps*())  
 ++winCount  
 }  
  
 *writeln*(`p=${winCount / n}`)  
}  
  
  
function *main*()  
{  
 let n = 1000000  
  
 *runSimulation*(n)  
}  
  
*main*()

**Sınıf Çalışması:** Bir tombala torbasında 1'den 99'a kadar numaralanmış (99 dahil) pullar bulunmaktadır. Bu tombala torbasıyla aşağıdaki oyunlar oynanmaktadır:

Çekilen bir pul torbaya geri atılmamak üzere

i) Torbadan 3 pul çekiliyor. Çekilen pulların toplamı 150' den küçük ise oyuncu kazanıyor.

ii) Torbadan 3 pul çekiliyor. Çekilen pulların toplamı asal sayı ise oyuncu kazanıyor.

iii) Torbadan 3 pul çekiliyor. En büyük değerli pul ile en küçük değerli pul arasındaki fark ortanca değerli puldan büyükse oyuncu kazanıyor.

Oynanacak her bir oyun için oyuncunun kazanma olasılığını en az 30000 oyunu simule ederek hesaplayınız!

**Çözüm:**

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *randomBoolean*()  
{  
 return *randomInt*(0, 2) === 1  
}  
  
function *randomInt*(min, max) *//[min, max)*{  
 return *parseInt*(*randomNumber*(min, max))  
}  
  
function *randomNumber*(min, max) *//[min, max)*{  
 return Math.random() \* (max - min) + min  
}  
  
function *isPrime*(val)  
{  
 if (val <= 1)  
 return false  
  
 if (val % 2 === 0)  
 return val === 2  
  
 if (val % 3 === 0)  
 return val === 3  
  
 if (val % 5 === 0)  
 return val === 5  
  
 if (val % 7 === 0)  
 return val === 7  
  
 for (let i = 11; i \* i <= val; i += 2)  
 if (val % i === 0)  
 return false  
  
 return true  
}  
  
  
function *mid*(a, b, c)  
{  
 if (a < b && b < c || c < b && b < a)  
 return b  
  
 if (b < a && a < c || c < a && a < b)  
 return a  
  
 return c  
}  
  
function *pullFirstNumber*()  
{  
 return *randomInt*(1, 100)  
}  
  
  
function *pullSecondNumber*(firstNumber)  
{  
 let secondNumber  
  
 do  
 secondNumber = *randomInt*(1, 100)  
 while (firstNumber === secondNumber)  
  
 return secondNumber  
}  
  
function *pullThirdNumber*(firstNumber, secondNumber)  
{  
 let thirdNumber  
  
 do  
 thirdNumber = *randomInt*(1, 100)  
 while (firstNumber === thirdNumber || secondNumber === thirdNumber)  
  
 return thirdNumber  
}  
  
let *sum* = (firstNumber, secondNumber, thirdNumber) => firstNumber + secondNumber + thirdNumber  
  
let *playFirstGame* = (firstNumber, secondNumber, thirdNumber) => *sum*(firstNumber, secondNumber, thirdNumber) < 150  
  
let *playSecondGame* = (firstNumber, secondNumber, thirdNumber) => *isPrime*(*sum*(firstNumber, secondNumber, thirdNumber))  
  
let *playThirdGame* = (firstNumber, secondNumber, thirdNumber) =>  
 Math.max(firstNumber, secondNumber, thirdNumber) - Math.min(firstNumber, secondNumber, thirdNumber) < *mid*(firstNumber, secondNumber, thirdNumber)  
  
function *runSimulation*(n)  
{  
 let firstGameWinCount = 0  
 let secondGameWinCount = 0  
 let thirdGameWinCount = 0  
  
 for (let i = 0; i < n; ++i) {  
 let firstNumber = *pullFirstNumber*()  
 let secondNumber = *pullSecondNumber*(firstNumber)  
 let thirdNumber = *pullThirdNumber*(firstNumber, secondNumber)  
  
 *//writeln(`${firstNumber} ${secondNumber} ${thirdNumber}`)* if (*playFirstGame*(firstNumber, secondNumber, thirdNumber))  
 ++firstGameWinCount  
  
 if (*playSecondGame*(firstNumber, secondNumber, thirdNumber))  
 ++secondGameWinCount  
  
 if (*playThirdGame*(firstNumber, secondNumber, thirdNumber))  
 ++thirdGameWinCount  
 }  
  
 let p1 = firstGameWinCount / n  
 let p2 = secondGameWinCount / n  
 let p3 = thirdGameWinCount / n  
  
  
 *writeln*(`p1=${p1}`)  
 *writeln*(`p2=${p2}`)  
 *writeln*(`p1=${p3}`)  
}  
  
  
  
function *main*()  
{  
 let n = 30000  
 *runSimulation*(n)  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı 1, 2 veya 3 basamaklı bir sayının Türkçe yazı karşılığını döndüren numToStrTR3D fonksiyonunu yazınız ve test ediniz. Fonksiyon basamak sayısı kontrolü yapmayacaktır.

**Çözüm:**

function *randomBoolean*()  
{  
 return *randomInt*(0, 2) === 1  
}  
  
function *randomInt*(min, max) *//[min, max)*{  
 return *parseInt*(*randomNumber*(min, max))  
}  
  
function *randomNumber*(min, max) *//[min, max)*{  
 return Math.random() \* (max - min) + min  
}  
  
let onesTR = ["", "bir", "iki", "üç", "dört", "beş", "altı", "yedi", "sekiz", "dokuz"]  
let tensTR = ["", "on", "yirmi", "otuz", "kırk", "elli", "altmış", "yetmiş", "seksen", "doksan"]  
  
function *numToStrTR3D*(val)  
{  
 if (val === 0)  
 return "sıfır"  
  
 let str = val > 0 ? "" : "eksi"  
 val = Math.abs(val)  
  
 let a = *parseInt*(val / 100)  
 let b = *parseInt*(val / 10) % 10  
 let c = val % 10  
  
 if (a !== 0) {  
 if (a !== 1)  
 str += onesTR[a]  
 str += "yüz"  
 }  
  
 if (b !== 0)  
 str += tensTR[b]  
  
 if (c !== 0)  
 str += onesTR[c]  
  
 return str  
}  
  
function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *runTestNumToStrTR3D*()  
{  
 *writeln*(`0->${*numToStrTR3D*(0)}`)  
  
 for (let i = 0; i < 20; ++i) {  
 let val = *randomInt*(-999, 1000)  
  
 *writeln*(`${val} -> ${*numToStrTR3D*(val)}`)  
 }  
}  
  
function *main*()  
{  
 *runTestNumToStrTR3D*()  
}  
  
*main*()

**Sınıf Çalışması:** Parametresi ile aldığı sayıyı 3'erli basamaklara ayırarak bir diziye yerleştiren ve dizinin referansını döndüren getDigitsInThrees metodunu yazınız ve test ediniz

1234567->1 234 567

1 -> 1

3456 -> 3 456

**Çözüm:**

function *randomBoolean*()  
{  
 return *randomInt*(0, 2) === 1  
}  
  
function *randomInt*(min, max) *//[min, max)*{  
 return *parseInt*(*randomNumber*(min, max))  
}  
  
function *randomNumber*(min, max) *//[min, max)*{  
 return Math.random() \* (max - min) + min  
}  
  
function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *getDigitsInThrees*(val)  
{  
 val = Math.abs(val)  
 let digits = []  
  
 while (val !== 0) {  
 digits.unshift(val % 1000)  
 val = *parseInt*(val / 1000)  
 }  
  
 return digits  
}  
  
function *main*()  
{  
 for (let i = 0; i < 30; ++i) {  
 let val = *randomInt*(-200000000, 200000000)  
 *writeln*(`val=${val}`)  
 *getDigitsInThrees*(val).forEach(val => *writeln*(val))  
 *writeln*("/////////////////")  
 }  
}  
  
*main*()

**Sınıflar**

ES ile nesne yönelimli programlama ES6 öncesinde de yapılabilmekteydi. Bunun için bir object ilk değer verme sentaksı ya da bir fonksiyon bildirimi kullanılabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *Product* = **function** (id, name, stock) {  
 **this**.**id** = id  
 **this**.**name** = name  
 **this**.**stock** = stock  
 **this**.toString = () => **"["** + **this**.**id** + **"]"** + **this**.**name** + **"-"** + **this**.**stock**  
}  
  
**function** *main*()  
{  
 **let** p = **new** *Product*(1234, **"laptop"**, 10)  
  
 *writeln*(p.toString());  
  
}  
  
*main*()

Örneğin bir nesne türünden dizi bildirimi de yapılabilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *Product* = **function** (id, name, stock) {  
 **this**.**id** = id  
 **this**.**name** = name  
 **this**.**stock** = stock  
 **this**.toString = () => **"["** + **this**.**id** + **"]"** + **this**.**name** + **"-"** + **this**.**stock**  
}  
  
**function** *main*()  
{  
 **let** products = [  
 **new** *Product*(1234, **"laptop"**, 10),  
 **new** *Product*(12345678, **"mouse"**, 11),  
 **new** *Product*(12345, **"klavye"**, 10)]  
  
 **for** (**let** p **of** products)  
 *writeln*(p.toString())  
}  
  
*main*()

Burada new operatörü programcının nesne bildiriminde kullanılan fonksiyon da this dönülmediği için zorunludur. Eğer nesne bildiriminde kullanılan fonksiyon this dönerse new operatörü kullanılmasa da olur. Fakat bu şekilde nesne yaratılması this dönülse bile her javascript yorumlayıcısında çalışmayabilir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**let** *Product* = **function** (id, name, stock) {  
 **this**.**id** = id;  
 **this**.**name** = name;  
 **this**.**stock** = stock;  
 **this**.toString = () => **"["** + **this**.**id** + **"]"** + **this**.**name** + **"-"** + **this**.**stock**;  
  
 **return this**;  
};  
  
**function** *main*()  
{  
 **let** products = [  
 **new** *Product*(1234, **"laptop"**, 10),  
 *Product*(12345678, **"mouse"**, 11),  
 *Product*(12345, **"klavye"**, 10)];  
  
 **for** (**let** p **of** products)  
 *writeln*(p.toString());  
}  
  
*main*()

ES ile object içerisindeki değişkenler iki tırnak ile de bildirilebilmektedir:

**function** *writeln*(a)  
{  
 ***console***.log(a)  
}  
  
**function** *toString*()  
{  
 **return `[**${**this**.**id**}**]**${**this**.**name**}**-**${**this**.**stock**}**`**}  
  
**function** *main*()  
{  
 **let** product = {  
 **"id"**: 12345,  
 **"name"**: **"laptop"**,  
 **"stock"**: 10,  
 **"toString"** : *toString* }  
  
 *writeln*(product.*toString*())  
}  
  
*main*()

ES6 ile birlikte artık bir sınıf bildirimi, türetme, çok biçimlilik gibi kavramlar da dile dahil edilmiştir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Product {  
 constructor(id, name, stock)  
 {  
 this.id = id;  
 this.name = name;  
 if (arguments.length >= 3)  
 this.stock = stock;  
 else  
 this.stock = 0;  
 }  
  
 toStr()  
 {  
 return "[" + this.id + "]" + this.name + "-" + this.stock;  
 }  
}  
  
  
function *main*()  
{  
 let p = new Product(1, 'keyboard', 10)  
  
 *writeln*(p.toStr())  
}  
  
*main*()

Burada Product sınıfı içerisinde bildirilen constructor (ctor) içerisinde aynı zamanda property elemanları da (sınıfın veri elemanları gözüyle de bakılabilir) tanımlanmıştır. Sınıflar içerisinde bildirilen metotlarda (fonksiyonlar) function anahtar sözcüğü kullanılmaz.

ES6 ile bir sınıf bildirimi içerisinde sınıfın elemanlarına erişim için get ve set propertyleri yazılabilir:  
  
function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Point {  
 constructor(x, y)  
 {  
 this.\_x = x  
 this.\_y = y  
 }  
  
 get x() { return this.\_x }  
  
 set x(value) { this.\_x = value}  
  
 get y { return this.\_y }  
  
 set y(value) { this.\_y = value}  
  
 toString()  
 {  
 return `(${this.\_x}, ${this.\_y})`  
 }  
}  
  
function *main*()  
{  
 let p = new Point(100, 100)  
  
 *writeln*(`x = ${p.x}, y = ${p.y}`)  
  
 p.x = 200  
  
 *writeln*(`x = ${p.x}, y = ${p.y}`)  
}  
  
*main*()

ES6 ile birlikte bir sınıfın static fonksiyonu da olabilmektedir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Point {  
 constructor(x, y)  
 {  
 this.\_x = x  
 this.\_y = y  
 }  
  
 static *distance*(x1, y1, x2, y2)  
 {  
 return Math.sqrt((x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2))  
 }  
  
 get x() { return this.\_x }  
  
 set x(value) { this.\_x = value}  
  
 get y() { return this.\_y }  
  
 set y(value) { this.\_y = value}  
  
 toString()  
 {  
 return `(${this.\_x}, ${this.\_y})`  
 }  
}  
  
function *main*()  
{  
 let x1 = 100  
 let y1 = 100  
 let x2 = 50  
 let y2 = 50  
  
 *writeln*(Point.*distance*(x1, y1, x2, y2))  
}  
  
*main*()

ES6 ile birlikte bir sınıf başka bir sınıftan da türetilebilmektedir (inheritance). Burada Java programlama dilindeki gibi extends anahtar sözcüğü kullanılmaktadır.

ES’ de türemiş sınıf ctor’u içerisinde taban sınıfın ctor’unu çağırmak programcının sorumluluğundadır. Programcı bunu super ctor sentaksı ile yapabilir. Bu sentaks ES’ye göre zorunludur. Bazı ES motorları super ctor sentaksı kullanılmadan da türetmeye izin verse de super ctor sentaksı mutlaka kullanılmalıdır. super anahtar sözcüğü taban sınıfın diğer fonksiyonlarına da ulaşmak için kullanılabilmektedir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class A {  
 constructor()  
 {  
 *writeln*("A constructor")  
 }  
 foo() { *writeln*("A foo")}  
}  
  
class B extends A {  
 constructor()  
 {  
 super()  
 *writeln*("B.constructor")  
 }  
 foo()  
 {  
 *writeln*("B.foo")  
 super.foo()  
 }  
}  
  
function *main*()  
{  
 let b = new B()  
}  
  
*main*()

Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class A {  
 constructor(val)  
 {  
 this.val = val  
 *writeln*("A constructor")  
 }  
 foo() { *writeln*("A foo")}  
}  
  
class B extends A {  
 constructor(val, a)  
 {  
 super(val)  
 this.a = a  
 *writeln*("B.constructor")  
 }  
 foo()  
 {  
 *writeln*("B.foo")  
 super.foo()  
 }  
}  
  
function *main*()  
{  
 let b = new B(10, 20)  
  
 *writeln*(`b.val=${b.val}, b.a=${b.a}`)  
}  
  
*main*()

Burada programcı türemiş sınıf içerisinde super ctor sentaksı taban sınıf bölümüne ilgili değer aktarmıştır. Aşağıdaki örnekte multiply metodu adeta “override” edilmiştir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class A {  
 constructor(val)  
 {  
 this.val = val  
 }  
  
 get val() {return this.\_val}  
 set val(value) {this.\_val = Math.abs(value)}  
  
 multiply(a)  
 {  
 return a \* this.\_val  
 }  
}  
  
class B extends A {  
 constructor(val)  
 {  
 super(val);  
 }  
  
 multiply(a) *//override* {  
 return super.multiply(a) \* a  
 }  
}  
  
  
function *main*()  
{  
 let a = new B(10)  
 *writeln*(a.multiply(20))  
}  
  
*main*()

Aşağıdaki örnekte basit bir Employee hiyararşisi ile çok biçimli gibi kullanılabilen bir örnek yapılmıştır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Employee {  
 constructor(name, citizenId, address)  
 {  
 this.\_name = name;  
 this.\_citizenId = citizenId  
 this.\_address = address  
 }  
   
 get name() {return this.\_name}  
 set name(str) {this.\_name = str}  
  
 get citizenId() {return this.\_citizenId}  
 set citizenId(id) {this.\_citizenId = id}  
  
 get address() {return this.\_address}  
 set address(addr) {this.\_address = addr}  
 calculatePayment() {return 0}  
}  
  
class HumanResources {  
 *//...* payInsurance(employee)  
 {  
 *writeln*(`name:${employee.name}`)  
 *writeln*(`citizenId:${employee.citizenId}`)  
 *writeln*(`address:${employee.address}`)  
 *writeln*(`payment:${employee.calculatePayment()}`)  
 }  
}  
  
class Worker extends Employee {  
 constructor(name, citizenId, address, feePerHour, hourPerDay)  
 {  
 super(name, citizenId, address);  
 this.\_feePerHour = feePerHour  
 this.\_hourPerDay = hourPerDay  
 }  
  
 get feePerHour()  
 {  
 return this.\_feePerHour;  
 }  
  
 set feePerHour(value)  
 {  
 this.\_feePerHour = value;  
 }  
  
 get hourPerDay()  
 {  
 return this.\_hourPerDay;  
 }  
  
 set hourPerDay(value)  
 {  
 this.\_hourPerDay = value;  
 }  
  
 calculatePayment()  
 {  
 return this.\_feePerHour \* this.\_hourPerDay \* 30  
 }  
}  
  
class ProjectWorker extends Worker {  
 constructor(name, citizenId, address, feePerHour, hourPerDay, extra)  
 {  
 super(name, citizenId, address, feePerHour, hourPerDay)  
 this.\_extra = extra  
 }  
  
 get extra()  
 {  
 return this.\_extra;  
 }  
  
 set extra(value)  
 {  
 this.\_extra = value;  
 }  
  
 calculatePayment()  
 {  
 return super.calculatePayment() + this.\_extra;  
 }  
}  
  
function *runCompanyApp*()  
{  
 let humanResources = new HumanResources()  
 let worker = new Worker("Ali", "12345678912", "Şişli", 78, 8)  
 let projectWorker = new ProjectWorker("Ali", "12345678912", "Şişli", 78, 8, 500)  
  
 humanResources.payInsurance(worker)  
 *writeln*("///////////")  
 humanResources.payInsurance(projectWorker)  
}  
  
function *main*()  
{  
 *runCompanyApp*()  
}  
  
*main*()

ES6 çoklu türetmeyi desteklemez.

Aşağıdaki örneği inceleyiniz:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Circle {  
 constructor(r = 0)  
 {  
 this.r = r  
 }  
  
 get r()  
 {  
 return this.\_r;  
 }  
  
 set r(value)  
 {  
 this.\_r = Math.abs(value)  
 }  
  
 get area() {return Math.PI \* this.\_r \* this.\_r}  
 get circumference() {return 2 \* Math.PI \* this.\_r}  
}  
  
class Point {  
 constructor(x = 0, y = 0)  
 {  
 this.\_x = x;  
 this.\_y = y;  
 }  
  
 get x()  
 {  
 return this.\_x;  
 }  
  
 set x(value)  
 {  
 this.\_x = value;  
 }  
  
 get y()  
 {  
 return this.\_y;  
 }  
  
 set y(value)  
 {  
 this.\_y = value;  
 }  
  
 static *distance*(x1, y1, x2, y2) {return Math.sqrt((x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2))}  
  
 distance(point) {return Point.*distance*(this.\_x, this.\_y, point.x, point.y)}  
  
 offset(dx, dy = dx)  
 {  
 this.\_x += dx  
 this.\_y += dy  
 }  
  
 toString() {return `{x: ${this.\_x}, y: ${this.\_y}}`}  
}  
  
class AnalyticalCircle extends Circle {  
 constructor(r = 0, x = 0, y = 0)  
 {  
 super(r)  
 this.\_center = new Point(x, y)  
 }  
  
 get center() {return new Point(this.x, this.y)}  
  
 set center(value) {this.setCenter(value.x, value.y)}  
  
 get x() {return this.\_center.x}  
  
 set x(value) {this.\_center.x = value}  
  
 get y() {return this.\_center.y}  
  
 set y(value) {this.\_center.y = value}  
  
 setCenter(x = 0, y = 0) {this.x = x; this.y = y}  
  
 offset(dx, dy = dx) {this.\_center.offset(dx, dy)}  
}  
  
function *main*()  
{  
 let ac = new AnalyticalCircle(-3.4, 100, 100)  
  
 let center = ac.center  
  
 center.offset(20, 30)  
  
 *writeln*(`{x : ${center.x}, y : ${center.y}}`)  
 *writeln*(`{x : ${ac.x}, y : ${ac.y}}`)  
  
 ac.offset(20, 89)  
  
 *writeln*(`{x : ${ac.x}, y : ${ac.y}}`)  
  
 ac.center = new Point(200, 200)  
  
 *writeln*(`{x : ${ac.x}, y : ${ac.y}}`)  
  
 *writeln*(new Point(3, 4).distance(new Point(5, 6)))  
}  
  
*main*()

***Anahtar Notlar:*** *ES6 ile birlikte bir sınıfa static fonksiyonlar eklenebildiği için bazı durumlarda mantıksal olarak bir araya toplamak için global fonksiyon yasmak yerine, static fonksiyonlar yazılabilir. Aşağıdaki örnekte tüm fonksiyonları static olan bir sınıf yazılmıştır:*

*function writeln(a)  
{  
 console.log(a)  
}  
  
function write(a)  
{  
 process.stdout.write(a)  
}  
  
class Random {  
 static nextNumber(min, max) //[min, max)  
 {  
 return Math.random() \* (max - min) + min  
 }  
  
 static nextInt(min, max) //[min, max)  
 {  
 return parseInt(Random.nextNumber(min, max))  
 }  
  
 static nextBoolean()  
 {  
 return Random.nextInt(0, 2) === 1  
 }  
}  
  
function main()  
{  
 for (let i = 0; i < 10; ++i)  
 write(Random.nextInt(1, 100) + " ")  
  
 writeln("\n////////////////////")  
  
 for (let i = 0; i < 10; ++i)  
 writeln(Random.nextNumber(3.4, 7.789))  
  
 writeln("////////////////////")  
  
 for (let i = 0; i < 10; ++i)  
 writeln(Random.nextBoolean())  
}  
  
main()*

**Hata Ayıklama (Error/Exception Handling)**

ES yorumlayıcı ile çalışan bir programlama dili olduğundan sentaks ve semantik hatalar da çalışma zamanı sırasında oluşur. Yani örneğin genel olarak akış sentaks ya da semantik olarak hatalı bir koda gelene kadar sentaks ve/veya semantik hata anlaşılamayabilir. Bu durum çok bariz hatalı durumlar için bazı yorumlayıcılar tarafından anlaşılabilse de programcı buna dikkat etmelidir. Üstelik yorumlayıcıdan yorumlayıcıya sentaks ve semantik hataların anlaşılması değişkenlik gösterebilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 100; ++i) {  
 let a = *parseInt*(Math.random() \* (max - min) + min)  
 *writeln*(`a=${a}`)  
 if (a) {  
 *writeln*("Doğru")  
 *//...* }  
 else {  
 foo("Test")  
 *//...* }  
 }  
}  
  
  
*main*()

Burada akış if deyiminin doğru kısmı çalıştırıldığı sürece foo isminin var olup olmadığının önemi yoktur. Akış else kısmına gelirse error oluşur.

Bu durumda ES için genel olarak 3(üç) çeşit hata vardır denebilir: Sentaks hataları (Syntax errors), Çalışma zamanı hataları (Runtime Errors) ve Mantıksal hatalar (Logical Errors):

Sentaks hataları dilin sözdizimine yönelik hatalardır. Bu hatalar ES’ de sadece hatanın olduğu akışı etkiler (thread). Eş zamanlı (asenkron) diğer akışlar bundan etkilenmez. Bu hatalar yakalanıp işlenecek hatalardan değildir.

Çalışma zamanı hataları programın çalışma zamanında olası hata durumlarını (exception) temsil eder. Programcının yakalayıp işleyebileceği hatalardandır. Çalışma zamanında oluşan durumların her zaman hata olması gerekmez. Yakalanmaması durumunda yalnızca hatanın meydana geldiği akış sonlanır.

Mantıksal hatalar yakalanması mümkün olmayan hatalardır. Santaks ya da çalışma zamanı hataları gibi değil, script içerisinde mantıksal yapılan bir hatalı durumda meydana gelir. Bu durum genelde kullanılan ortamın iyi bilinmemesinden kaynaklanır.

ES’ de hata ayıklama try, catch, throw ve finally anahtar sözcükleri ile yapılır. ES’de herhangi bir tür fırlatılabileceği gibi genel olarak çalışma zamanı sırasında Error sınıfı türünden nesneler fırlatılır. ES’ de Error sınıfı dışında fırlatılabilecek bazı hata nesneleri de bulunur:

* EvalError: Bu hata nesnesi eval isimli global fonksiyon tarafından fırlatılır eval fonksiyonu parametresi ile aldığı yazıyı ES olarak yorumlayıp çalıştırır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 try {  
 let expression = 'writeln("Merhaba"); console.log("nasılsın")'  
  
 *eval*(expression)  
 }  
 catch (e) {  
 *writeln*(e.toString())  
 }  
}  
  
  
*main*()

Burada eval fonskiyonuna verilen kod parçasında hata oluştuğunda yakalanabilir

* RangeError: Bu hata genel olarak numeric bir değişkenin sınır değerleri dışında değer aldığında fırlatılır.
* ReferenceError: Bu hata genel olarak geçersiz bir referansa erişim sağlandığında oluşur.
* SyntaxError: Herhangi bir sentaks hatası oluştuğunda fırlatılır. Çoğu yorumlayıcı yakalanmasına izin vermez
* TypeError: Bir değişkenin kullanımında türün hatalı olması durumunda fırlatılır:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 try {  
 let a = "ankara"  
  
 a.toUpperCase()  
 a.foo()  
 }  
 catch (e) {  
 *writeln*(e.toString())  
 }  
}  
  
  
*main*()

Bu sınıflar dışında kullanılan ortama göre başka hata nesneleri de olabilmektedir.

Bir hata throw anahtar sözcüğü ile aşağıdaki şekillerde fırlatılabilir:

* throw new Error([message])
* throw([message])
* throw new <sınıf ismi>([message])

Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class MathUtil {  
 static *log*(value)  
 {  
 if (value < 0)  
 throw new Error("Belirsiz")  
  
 if (value === 0)  
 throw new Error("Tanımsız")  
  
 return Math.log(value)  
 }  
}  
  
function *main*()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 100; ++i) {  
 try {  
 let result = MathUtil.*log*(Math.random() \* (max - min) + min)  
  
 *writeln*(result)  
 }   
 catch (ex) {  
 *writeln*(ex.message)  
 }  
 }  
}  
  
  
*main*()

Ya da örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class MathUtil {  
 static *log*(value)  
 {  
 if (value < 0)  
 throw("Belirsiz")  
  
 if (value === 0)  
 throw("Tanımsız")  
  
 return Math.log(value)  
 }  
}  
  
function *main*()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 100; ++i) {  
 try {  
 let result = MathUtil.*log*(Math.random() \* (max - min) + min)  
  
 *writeln*(result)  
 }  
 catch (ex) {  
 *writeln*(ex.toString())  
 }  
 }  
}  
  
  
*main*()

Programcı isterse Error sınıfından türetme yaparak kendi hata sınıfını da yazabilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class MathError extends Error {  
 constructor(message, errorCode)  
 {  
 super(message)  
 this.\_errorCode = errorCode  
 }  
  
 get errorMessage() {return `Message:${this.message}, Error Code:${this.\_errorCode}`}  
 get errorCode() {return this.\_errorCode}  
}  
  
class MathUtil {  
 static *log*(value)  
 {  
 if (value < 0)  
 throw new MathError("Belirsiz", -1)  
  
 if (value === 0)  
 throw new MathError("Tanımsız", 0)  
  
 return Math.log(value)  
 }  
}  
  
function *main*()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 100; ++i) {  
 try {  
 let result = MathUtil.*log*(Math.random() \* (max - min) + min)  
  
 *writeln*(result)  
 }  
 catch (ex) {  
 *writeln*(ex.errorMessage)  
 }  
 }  
}  
  
  
*main*()

ES’ de de akış error bakımından ele alınacaksa (handling) try bloğu içerisinde olmalıdır. try bloğu bir catch bloğu ve finally bloğu veya tek başına catch veya finally bloğu ile devam edebilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class MathError extends Error {  
 constructor(message, errCode)  
 {  
 super(message)  
 this.\_errCode = errCode  
 }  
  
 get errCode() {return this.\_errCode}  
}  
  
class MathUtil {  
 static *log*(value)  
 {  
 if (value < 0)  
 throw new MathError("Belirsiz", -1)  
  
 if (value === 0)  
 throw new MathError("Tanımsız", 0)  
  
 return Math.log(value)  
 }  
}  
  
function *main*()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 100; ++i) {  
 try {  
 let result = MathUtil.*log*(Math.random() \* (max - min) + min)  
  
 *writeln*(result)  
 }  
 catch (ex) {  
 *writeln*(`Message:${ex.message}, Error Code:${ex.errCode}`)  
 }  
 finally {  
 *writeln*("finally")  
 }  
 *writeln*("/////////////")  
 }  
}  
  
*main*()

try-finally kullanımı:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class MathError extends Error {  
 constructor(message, errCode)  
 {  
 super(message)  
 this.\_errCode = errCode  
 }  
  
 get errCode() {return this.\_errCode}  
}  
  
class MathUtil {  
 static *log*(value)  
 {  
 if (value < 0)  
 throw new MathError("Belirsiz", -1)  
  
 if (value === 0)  
 throw new MathError("Tanımsız", 0)  
  
 return Math.log(value)  
 }  
}  
  
function *doWork*(min, max)  
{  
 try {  
 let result = MathUtil.*log*(Math.random() \* (max - min) + min)  
  
 *writeln*(result)  
 }  
 finally {  
 *writeln*("doWork:finally")  
 }  
 *writeln*("/////////////")  
}  
  
function *main*()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 100; ++i) {  
 try {  
 *doWork*(min, max)  
 }  
 catch (ex) {  
 *writeln*(`main:Message:${ex.message}, Error Code:${ex.errCode}`)  
 }  
 finally {  
 *writeln*("main:finally")  
 }  
 *writeln*("/////////////")  
 }  
}  
  
*main*()

try bloklarının geri kalan çalışma sistematiği Java ile çok benzerdir.

***Anahtar Notlar:*** *Bilindiği gibi Java’ da exception sınıfları kategori checked ve unchecked olmak üzere iki gruba ayrılır. ES’ de böyle bir durum yoktur.*

Çok fazla exception handling yapılarak yazılan kodlarda çok fazla try bloğu olması kodun okunabilirliğini etkileyebilir. Bu durumda aşağıdaki gibi fonksiyonlar yazılarak kod karmaşıklığı engellenebilir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
  
function *subscribe*(action, errorAction, finallyAction)  
{  
 try {  
 return action()  
 }  
 catch (ex) {  
 errorAction(ex)  
 }  
 finally {  
 finallyAction()  
 }  
}  
  
function *subscribeForFinally*(action, finallyAction)  
{  
 try {  
 return action()  
 }  
 finally {  
 finallyAction()  
 }  
}  
  
  
class MathError extends Error {  
 constructor(message, errCode)  
 {  
 super(message)  
 this.\_errCode = errCode  
 }  
  
 get errCode() {return this.\_errCode}  
}  
  
class MathUtil {  
 static *log*(value)  
 {  
 if (value < 0)  
 throw new MathError("Belirsiz", -1)  
  
 if (value === 0)  
 throw new MathError("Tanımsız", 0)  
  
 return Math.log(value)  
 }  
}  
  
function *doWork*(min, max)  
{  
 *subscribe*(() => *writeln*(MathUtil.*log*(Math.random() \* (max - min) + min)), () => *writeln*("doWork:finally"))  
  
 *writeln*("/////////////")  
}  
  
function *main*()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 100; ++i) {  
 *subscribe*(() => *doWork*(min, max),  
 ex => *writeln*(`main:Message:${ex.message}, Error Code:${ex.errCode}`), () => *writeln*("main:finally"))  
 *writeln*("/////////////")  
 }  
}  
  
*main*()

Burada subscribe ve subscribeForFinally fonksiyonları try bloğunda her durumda ne yapılacağını callback olarak almaktadır.

**Türlerin prototype elemanı**

Her türün prototype denilen bir elemanı vardır. Bunun sayesinde bir türe yeni bir eleman eklenebilmektedir:

Aşağıdaki örnekte s referansının gösterdiği nesneye count isimli bir metot eklenmiştir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Sample {  
 constructor(value)  
 {  
 this.\_value = value  
 }  
  
 get value() {return this.\_value}  
}  
  
function *main*()  
{  
 let s = new Sample(10000)  
  
 s.count = function () {  
 return this.\_value === 0 ? 1 : *parseInt*(Math.log10(Math.abs(this.\_value))) + 1  
 }  
  
 *writeln*(`Number of digits:${s.count()}`)  
  
 let k = new Sample(123)  
  
 *writeln*(k.count()) *//error*}  
  
*main*()

prototype elemanı ile Sample sınıfına aşağıdaki gibi count metodu eklenmiştir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Sample {  
 constructor(value)  
 {  
 this.\_value = value  
 }  
  
 get value() {return this.\_value}  
}  
function *main*()  
{  
 let s = new Sample(10000)  
  
 Sample.prototype.count = function () {  
 return this.\_value == 0 ? 1 : *parseInt*(Math.log10(Math.abs(this.\_value)) + 1)  
 }  
  
 *writeln*(s.count())  
  
 let k = new Sample(123)  
  
 *writeln*(k.count())  
}  
  
*main*()

Örneğin:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
class Sample {  
 constructor(value)  
 {  
 this.\_value = value  
 }  
  
 get value() {return this.\_value}  
}  
  
function *main*()  
{  
 let s = new Sample(10000)  
  
 Sample.prototype.count = function () {  
 return this.\_value === 0 ? 1 : *parseInt*(Math.log10(Math.abs(this.\_value))) + 1  
 }  
  
 *writeln*(`Number of digits:${s.count()}`)  
  
 let k = new Sample(123)  
  
 *writeln*(`Number of digits:${k.count()}`)  
}  
  
*main*()

Aşağıdaki örnekte Number sınıfına count isimli fonksiyon çalışma zamanı sırasında eklenmiştir:

function *writeln*(a)  
{  
 console.log(a)  
}  
  
function *main*()  
{  
 let val = *parseInt*(Math.random() \* 10000)  
  
 *writeln*(`val=${val}`)  
  
 Number.prototype.count = function () {  
 return this === 0 ? 1 : *parseInt*(Math.log10(Math.abs(this))) + 1  
 }  
  
 *writeln*(`Number of digits:${val.count()}`)  
}  
  
*main*()

**Modüller**

Bir js dosyasına modül denir. ES6’dan önce ES’de bir js dosyasından başka bir js dosyasındaki bir ismi kullanmanın doğrudan bir yolu yoktu. Bu sebeple bazı ES kullanan teknolojiler bu işlemi yapabilecek araçları da barındırıyorlardı. Örneğin NodeJS teknolojisinde farklı modüller ile çalışmak ES6 olmasa da mümkündür.

***Anahtar Notlar:*** *ES6’ ya bazı durumlarda çıktığı yıl olan 2015 dolayısıyla ES2015 de denilmektedir. Örneğin NodeJS terminolojisinde modüller iki gruba ayrılır: CommonJS moduller ya da NodeJS moduller, ES2015 veya ES6 modüller.*

***Anahtar Notlar:*** *ES2015 ile birlikte modül dosyalarının NodeJS gibi farklı modüllerle kendine özgü çalışma yöntemi olan teknolojilerde ES6 modüllerinin kullanımlarında uzantılarının “.mjs” biçiminde seçilmesi gerekebilir.*

***Anahtar Notlar:*** *ES6 ile**birlikte bazı durumların kısıtlı bir biçimde kullanılmasını veya doğrudan kullanılamaması gibi kuralların devreye girmesine “strict mode” denilmektedir. ES6’nın ilk zamanlarında bu şekilde bildiriler gerekebiliyordu. Ancak şu an ES6’dan sonraki sürümler yaygın olarak kullanıldığında bu bildirime çoğu zaman gerek yoktur. Zaten kısıtlar vardır ve kuralları da belirlidir.*

Bir modül içerisinde kullanılan bir isme başka bir modülden doğrudan erişilemez. Bunun için modül içerisinde bir isim export anahtar sözcüğü ile bildirilmelidir. export anahtar sözcüğün iki kullanımı vardır: isimlerin export edilmesi (named exports), varsayılan export (default export).

**Modül içerisinden export edilmesi (named exports)**

Bu kullanımın genel biçimi:

export <isim>

veya

export {<isim listesi>}

export function *writeln*(msg)  
{  
 console.log(msg)  
}

export let x = 10

Yukarıdaki export değişkenler aynı script içerisinde ise aşağıdaki gibi bir export bildirimi yapılabilir:

function *writeln*(msg)  
{  
 console.log(msg)  
}  
  
let x = 10  
  
export {*writeln*, x}

Aşağıdaki örnekte bir sınıf export edilmiştir:

export class Random {  
 static *nextDouble*(min, max) *//[min, max)* {  
 if (min >= max)  
 throw new Error("min must be less than max")  
  
 return Math.random() \* (max - min) + min  
 }  
  
 static *nextInt*(min, max) *//[min, max)* {  
 if (min >= max)  
 throw new Error("min must be less than max")  
  
 return Math.floor(Math.random() \* (max - min) + min)  
 }  
  
 static *nextBoolean*()  
 {  
 return this.*nextInt*(0, 2) === 1;  
 }  
}

**Varsayılan export**

Bir modülün ismi zaten export edilmiştir. Ancak programcı isterse bir modülü default bir isimle de export edebilir. Bir modül içerisinde bir tane default yapılabilir:

export default *writeln*

**Modüllerin import edilmesi**

Bir modül içerisinde başka bir modülde export edilen bir elemanı kullanabilmek için import anahtar sözcüğü kullanılır. Bir modül içerisinde birden fazla import bildirimi olabilir. Bildirim sırasının önemi yoktur. import bildirimleri iki şekilde kullanılabilir: isimlerin import edilmesi, default export edilen modüllerin import edilmesi

İsimlerin import edilmesinin genel biçimi:

import {isim listesi} from <modül ismi>

Örneğin:

import {*writeln*} from "./util.mjs"  
import {Random} from "./Random.mjs"

Tüm isimler aşağıdaki gibi yıldız (\*) (asterisk operator) ile takma isim (alias) kulanarak aşağıdaki gibi bildirilebilir:

import \* as util from "./util.mjs"

import bildirimde {} içerisindeki isimlere de takma isim verilebilir:

import {Random as R} from "./Random.mjs"

import bildiriminde takma ismin bir kullanımı:

import {*writeLine*} from "./consoleutil.mjs";  
import {*writeLine* as writeLog} from "./logutil.mjs";  
import {*randomInt*} from "./randomutil.mjs";  
  
function log(value)  
{  
 if (value < 0)  
 throw new Error("Belirsiz")  
  
 if (value === 0)  
 throw new Error("Tanımsız")  
  
 return Math.log(value)  
}  
  
function main()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 10; ++i) {  
 try {  
 let value = log(*randomInt*(min, max));  
  
 *writeLine*(`log(${value})=${value}`)  
 }  
 catch (ex) {  
 *writeLog*("Durum", ex.message)  
 }  
  
 *writeLine*("////////////////////")  
 }  
}  
  
main()

Aşağıdaki modülde iki tane isEven fonksiyonu export edilmiştir:

function *isPrime*(val)  
{  
 if (val <= 1)  
 return false  
  
 if (val % 2 == 0)  
 return val == 2  
  
 if (val % 3 == 0)  
 return val == 3  
  
 if (val % 5 == 0)  
 return val == 5  
  
 if (val % 7 == 0)  
 return val == 7  
  
 for (let i = 11; i \* i <= val; i += 2)  
 if (val % i == 0)  
 return false  
  
 return true  
}  
  
function *isEven*(value)  
{  
 console.log("isEven")  
 return value % 2 === 0  
}  
  
Number.prototype.isEven = function () {  
 return this % 2 === 0  
}  
  
export {*isPrime*, *isEven*}

Bu durumda isEven fonksiyonunu import eden modülde çağırma biçimine göre hangisinin çağrılacağına karar verilecektir:

import {*writeLine*} from "./consoleutil.mjs";  
  
import {*isEven*} from "./numberextensions.mjs";  
  
function main()  
{  
 let val = 10  
  
 *writeLine*(val.isEven())  
 *writeLine*(*isEven*(val))  
}  
  
main()

Örnek modüller ve kullanımları:

//consoleutil.js

function *write*(a)  
{  
 process.stdout.write(a)  
}  
  
function *writeLine*(a)  
{  
 *write*(`${a}\n`)  
}  
  
  
export {*writeLine*, *write*}

//logutil.js

function *writeLine*(message, a)  
{  
 console.log(`${message}:${a}`)  
}  
  
  
export {*writeLine*}

//exceptionutil.js

function *subscribe*(action, errorAction, finallyAction)  
{  
 try {  
 return action()  
 }  
 catch (ex) {  
 errorAction(ex)  
 }  
 finally {  
 if (arguments.length > 2)  
 finallyAction()  
 }  
}  
  
function *subscribeForFinally*(action, finallyAction)  
{  
 try {  
 return action()  
 }  
 finally {  
 finallyAction()  
 }  
}  
  
export {*subscribe*, *subscribeForFinally*}

//randomutil.js

function *randomDouble*(min, max) *//[min, max)*{  
 if (min >= max)  
 throw new Error("min must be less than max")  
  
 return Math.random() \* (max - min) + min  
}  
  
function *randomInt*(min, max) *//[min, max)*{  
 if (min >= max)  
 throw new Error("min must be less than max")  
  
 return *parseInt*(Math.random() \* (max - min) + min)  
}  
  
function *randomBoolean*()  
{  
 return this.*nextInt*(0, 2) === 1;  
}  
  
export {*randomInt*, *randomBoolean*, *randomDouble*}

//sample.js

import {*writeLine*} from "./consoleutil.mjs";  
import {*writeLine* as writeErr} from "./logutil.mjs";  
import {*subscribe*} from "./exceptionUtil.mjs";  
import {*randomInt*} from "./randomutil.mjs";  
  
function log(value)  
{  
 if (value < 0)  
 throw new Error("Belirsiz")  
  
 if (value === 0)  
 throw new Error("Tanımsız")  
  
 return Math.log(value)  
}  
  
function logOperationCallback(min, max)  
{  
 let value = log(*randomInt*(min, max));  
  
 *writeLine*(`log(${value})=${value}`)  
}  
  
function main()  
{  
 let min = -10  
 let max = 10  
  
 for (let i = 0; i < 10; ++i) {  
 *subscribe*(() => logOperationCallback(min, max), ex => *writeErr*("Durum", ex.message))  
 *writeLine*("////////////////////")  
 }  
}  
  
main()

**const değişkenler:**

const değişkenler const anahtar sözcüğü ile bildirilirler:

import {*writeLine*} from "./consoleutil.mjs";  
  
function main()  
{  
 const AVOGADRO\_NUMBER = 6.02E+23  
   
 *writeLine*(AVOGADRO\_NUMBER)  
}  
  
main()

const değişkenler faaliyet alanı bakımından let değişkenler ile aynıdır:

import {*writeLine*} from "./consoleutil.mjs";  
  
function main()  
{  
 {  
 const AVOGADRO\_NUMBER = 6.02E+23  
 }  
  
 *writeLine*(AVOGADRO\_NUMBER) *//error*}  
  
main()

const bir değişkene bir kez atama yapılabilir:

import {*writeLine*} from "./consoleutil.mjs";  
  
function main()  
{  
 const AVOGADRO\_NUMBER = 6.02E+23  
  
 AVOGADRO\_NUMBER = 6.02 *//error  
  
  
 writeLine*(AVOGADRO\_NUMBER)  
}  
  
main()

const değişkenlere atama bildirim sırasında yapılmalıdır:

import {*writeLine*} from "./consoleutil.mjs";  
  
function main()  
{  
 const AVOGADRO\_NUMBER *//error* AVOGADRO\_NUMBER = 6.02E+23  
  
  
 *writeLine*(AVOGADRO\_NUMBER)  
}  
  
main()

const değişkene ikinci defa atama yapılamayacağına dikkat ediniz, const bir referansın gösterdiği nesnenin içeriği değiştirilebilir:

import {*writeLine*} from "./consoleutil.mjs"  
  
class Sample {  
 constructor(val)  
 {  
 this.\_value = val  
 }  
  
  
 get val()  
 {  
 return this.\_value;  
 }  
  
 set val(value)  
 {  
 this.\_value = value;  
 }  
}  
  
  
function main()  
{  
 const sample = new Sample(10)  
  
 *writeLine*(sample.val)  
  
 ++sample.val  
  
 *writeLine*(sample.val)  
}  
  
main()

Benzer durum diziler için de geçerlidir:

import {*randomIntArray*, *displayIntArray*, *multiplyBy*} from "./arrayUtil.mjs"  
  
function main()  
{  
 const min = 10  
 const max = 100  
  
 const a = *randomIntArray*(10, min, max)  
  
 a.displayIntArray()  
 a.multiplyBy(2)  
 a.displayIntArray()  
}  
  
main()

import {*randomInt*} from "./randomutil.mjs";  
import {*writeLine*, *write*} from "./consoleutil.mjs";  
  
function *randomIntArray*(n, min, max)  
{  
 const a = []  
  
 for (let i = 0; i < n; ++i)  
 a.push(*randomInt*(min, max))  
  
 return a  
}  
  
function *displayIntArray*()  
{  
 for (let val of this)  
 *write*(`${val} `)  
  
 *writeLine*()  
}  
  
function *multiplyBy*(value)  
{  
 for (let i = 0; i < this.length; ++i)  
 this[i] \*= value  
}  
  
function *displayDoubleArray*()  
{  
 for (let val of this)  
 *writeLine*(val)  
}  
  
Array.prototype.displayIntArray = *displayIntArray*Array.prototype.displayDoubleArray = *displayDoubleArray*Array.prototype.multiplyBy = *multiplyBy*export {*randomIntArray*, *multiplyBy*, *displayIntArray*, *displayDoubleArray*}

İkinci defa atama yapılmayacak ve bildirim sırasında atama yapılabilecek her değişken const yapılmalıdır. Bu durum yorumlayıcıların da efektif olarak kod çalıştırmasını sağlar. Ayrıca bir değişkenin const yapılması, let bir değişkenin okunabilirliğini artırır. Bir değişkenin let olması “o değişkene faaliyet alanı boyunca atama yapılacağı anlamına gelir. Çünkü atama yapılmıyorsa programcı zaten bunu const yapar” şeklinde anlaşılır.

**Destructing assignments:**

Parçalama ataması ES’de bir dizinin elemanlarının ve bir nesnenin property değerlerlerinin kolay bir biçimde değişkenlere atanmasını sağlar. ES6 ile eklenmiştir:

import {*writeLine*} from "./consoleutil.mjs";  
import {Point} from "./Point.mjs";  
  
function main()  
{  
 const point = new Point(100, 200)  
  
 const {x, y} = point  
  
 *writeLine*(`x = ${x}, y = ${y}`)  
}  
  
main()

Burada point referansının göstediği nesnenin parçaları x ve y const değişkenlerine atanmıştır. Parçalanan değişkenlerin isimleri nesnenin property elemanlarının isimleri ile aynı olmalıdır. Değişkenlerin isimleri aşağıdaki gibi değiştirilebilir:

import {*writeLine*} from "./consoleutil.mjs";  
import {Point} from "./Point.mjs";  
  
function main()  
{  
 const point = new Point(100, 200)  
  
 const {x: a, y: b} = point  
  
 *writeLine*(`x = ${a}, y = ${b}`)  
}  
  
main()

Örneğin:

import {*writeLine*} from "./consoleutil.mjs";  
import {Point} from "./Point.mjs";  
  
function main()  
{  
 const point = new Point(100, 200)  
  
 let {x, y} = point  
  
 *writeLine*(`x = ${x}, y = ${y}`)  
}  
  
main()

Diziye parçalama sırasında [] kullanılmalıdır. Dizinin bazı elemanları parçalamada istenmezse virgül koyarak elaman es geçilebilir:

import {*writeLine*} from "./consoleutil.mjs";  
  
function main()  
{  
 const numbers = [1, 2, 3]  
  
 const [x, , z] = numbers  
  
 *writeLine*(`x = ${x}, z = ${z}`)  
}  
  
main()

Örneğin:

import {*writeLine*} from "./consoleutil.mjs";  
import {Point} from "./Point.mjs";  
  
function createPoint(x, y)  
{  
 *//...* return new Point(x, y)  
}  
  
  
function main()  
{  
 const {x, y} = createPoint(100, 200)  
  
 *writeLine*(`x = ${x}, y = ${y}`)  
}  
  
main()

Örneğin:

import {*writeLine*} from "./consoleutil.mjs";  
import {Point} from "./Point.mjs";  
  
function createPoint(x, y)  
{  
 *//...* return new Point(x, y)  
}  
  
  
function main()  
{  
 const points = [new Point(100, 200), new Point(20, 45), new Point(50, -70)]  
  
 for (const {x, y} of points)  
 *writeLine*(`x = ${x}, y = ${y}`)  
}  
  
main()

Örneğin:

import {*writeLine*} from "./consoleutil.mjs";  
  
class Person {  
 constructor(name, address)  
 {  
 this.\_name = name  
 this.\_address = address  
 }  
  
 get name() {  
 return this.\_name;  
 }  
  
 set name(value) {  
 this.\_name = value;  
 }  
  
 get address() {  
 return this.\_address;  
 }  
  
 set address(value) {  
 this.\_address = value;  
 }  
}  
  
function main()  
{  
 const people = [new Person("Ali", "şişli"), new Person("Oğuz", "Göktürk")]  
  
 for (const {name: n, address: a} of people)  
 *writeLine*(`name = ${n}, address = ${a}`)  
}  
  
main()

**Bazı yararlı global fonksiyonlar**

ES’ de NodeJS gibi teknolojilerde de çok sık kullanılan bazı global fonksiyonlar bulunur.

**setTimeout Fonksiyonu**

Bu fonksiyon verilen bir süre sonunda verilen bir arkaplan (callback) fonksiyonun çağrılmasını sağlar. Süre milisaniye cinsinden verilir:

import {*writeLine*} from "./consoleutil.mjs";  
  
function main()  
{  
 *setTimeout*(() => *writeLine*("Birinci timeout süresi doldu"), 3000)  
 *writeLine*("Arka planda akış süre dolmasını bekliyor")  
 *setTimeout*(() => *writeLine*("İkinci timeout süresi doldu"), 5000)  
 *//...*}  
  
main()

Bu işlem asenkron yapılmaktadır. Yani varolan akış devam eder. Timeout ayrı bir akış olarak çalışmaktadır.

Set edilmiş olan bir timeout clearTimeout isimli bir fonksiyon ile devre dışı bırakılabilir:

import {*writeLine*} from "./consoleutil.mjs"  
  
function main()  
{  
 let timeout = *setTimeout*(() => *writeLine*("Süre doldu"), 4000)  
  
 let ms = *parseInt*(Math.random() \* 10000)  
  
 *writeLine*(ms)  
  
 *setTimeout*(() => *clearTimeout*(timeout), ms)  
  
 *writeLine*("main bitti")  
}  
  
main()

ES’de setInterval fonksiyonu ile arka planda periodic olarak çalışan bir akış (timer) oluşturulabilir:

import {*write*, *writeLine*} from "./consoleutil.mjs"  
  
function main()  
{  
 *setInterval*(() => *write*("."), 1000)  
 *writeLine*("main bitti")  
}  
  
main()

Aşağıdaki porgramı çalıştırınız:

import {*write*, *writeLine*} from "./consoleutil.mjs"  
  
function main()  
{  
 *setInterval*(() => *write*("?"), 1)  
 *setInterval*(() => *write*(":"), 1)  
 *writeLine*("main bitti")  
}  
  
main()

Set edilmiş olan bir interval clearInterval isimli bir fonksiyon ile iptal edilebilir:

import {*write*, *writeLine*} from "./consoleutil.mjs"  
import {*randomInt*} from "./randomutil.mjs";  
  
function main()  
{  
 let interval = *setInterval*(() => *write*("."), 1000)  
  
 let ms = *randomInt*(5000, 10000)  
  
 *writeLine*(`Süre:${ms}`)  
  
 *setTimeout*(() => *clearInterval*(interval), ms)  
  
 *writeLine*("main bitti")  
}  
  
main()

**Sınıf Çalışması:** Saniyede bir çalışan bir interval içerisinde rasgele sayılar üretilsin. 3 tane asal sayı geldiğinde interval’ı durduran program yazınız.

import {*isPrime*} from "./numberextensions.mjs";  
import {*write*, *writeLine*} from "./consoleutil.mjs";  
import {Random} from "./Random.mjs";  
  
let primeCount = 0  
let count = 0  
let randomGeneratorInterval = *setInterval*(randomGeneratorIntervalCallback, 1000)  
  
function randomGeneratorIntervalCallback()  
{  
 let val = Random.*nextInt*(1, 100)  
  
 ++count  
  
 if (val.isPrime()) {  
 *write*(`${val} `)  
 ++primeCount  
 }  
  
 if (primeCount === 3) {  
 *clearInterval*(randomGeneratorInterval)  
 *writeLine*(`\nCount:${count}`)  
 }  
}

**Promise Sınıfı**

Promise sınıfı, özellikle asenkron işlemleri daha kolay ve hata yapma olasılığını azaltacak biçimde yapmak için tasarlanmış standard bir sınıftır. Bir promise nesnesi yaratıldığında callback bir fonksiyon ister. Bu fonksiyonun iki tane callback fonksiyon alan parametresi bulunmaktadır. Genel olarak bu fonksiyonu çağırmayız. Bu fonksiyona geçilen fonksiyonları çağırırız. Promise referansını elde ettikten sonra programcı then fonksiyonu ile başarılı durumu, catch fonksiyonu ile başarısızlık durumunu asenkron olarak ele alabilir:

import {*randomBoolean*} from "./randomutil.mjs";  
import {*writeLine*} from "./consoleutil.mjs";  
  
function getMessage()  
{  
 let status = *randomBoolean*()  
  
 return new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 if (status)  
 resolve("Sözümü tuttum")  
  
 reject("Sözümü tutamadım")  
 }, 3000)  
 })  
}  
  
  
function main()  
{  
 let promise = getMessage()  
  
 promise.then(message => *writeLine*(message)).catch(message => *writeLine*(message))  
  
 *writeLine*("main bitti")  
}  
  
main()

Burada promise referansının then metodu ile başarılı durum (yani resolve ile çağrılan durum), catch metodu ile başarısız durum (yani reject ile çağrılan) asenkron olarak ele alınmıştır. Bir promise nesnesi bir t anında aşağıdaki durumlardan yalnızca birinde olabilir:

1. Fullfilled: Promise nesnesinin başarılı durumudur. Yani resolve çağrıldığı durumdur.
2. Rejected: Promise nesnesinin başarızılık durumudur. Yani reject fonksiyonun çağrıldığı durumdur.
3. Pending: Promise nesnesinin henüz daha işi yapıyor olduğu yani resolve veya reject’in henüz daha çağrılmadığı durumdur
4. Settled: Başarılı ve başarısız olma durumunun tamamlanmış olduğu durumdur.

Promise nesneleri zincirleme olarak (promise chain) kullanılabilir:

import {*randomInt*} from "./randomutil.mjs";  
import {*writeLine*} from "./consoleutil.mjs";  
  
class MessageInfo {  
 constructor(val, message)  
 {  
 this.\_val = val  
 this.\_message = message  
 }  
  
 get val()  
 {  
 return this.\_val;  
 }  
  
 set val(value)  
 {  
 this.\_val = value;  
 }  
  
 get message()  
 {  
 return this.\_message;  
 }  
  
 set message(value)  
 {  
 this.\_message = value;  
 }  
}  
  
function getMessage(messageInfo)  
{  
 return new Promise(resolve => resolve(`Value:${messageInfo.val}, Message:${messageInfo.message}`))  
}  
  
function getMessageInfo()  
{  
 let val = *randomInt*(-99, 99)  
  
 return new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 *writeLine*(`val=${val}`)  
 if (val > 0)  
 resolve(new MessageInfo(val, "Sözümü tuttum"))  
  
 reject(new MessageInfo(val, "Sözümü tutamadım"))  
 }, 3000)  
 })  
}  
  
function main()  
{  
 let promise = getMessageInfo()  
  
 promise  
 .then(getMessage)  
 .then(message => *writeLine*(message))  
 .catch(messageInfo => *writeLine*(messageInfo.message))  
}  
  
main()

Burada getMessageInfo fonksiyonunun geri dönüş değerinde elde edilen Promise referansı getMessage fonksiyonunun geri dönüş değerinden elde edilen Promise ile zincir biçiminde bağlanmıştır. Yukarıdaki örnekte getMessage fonksiyonunda Promise sınıfının resolve static fonksiyonu çağrılarak doğrudan Promise referansı döndürülebilir:

import {*randomInt*} from "./randomutil.mjs";  
import {*writeLine*} from "./consoleutil.mjs";  
  
class MessageInfo {  
 constructor(val, message)  
 {  
 this.\_val = val  
 this.\_message = message  
 }  
  
 get val()  
 {  
 return this.\_val;  
 }  
  
 set val(value)  
 {  
 this.\_val = value;  
 }  
  
 get message()  
 {  
 return this.\_message;  
 }  
  
 set message(value)  
 {  
 this.\_message = value;  
 }  
}  
  
function getMessage(messageInfo)  
{  
 return Promise.resolve(`Value:${messageInfo.val}, Message:${messageInfo.message}`)  
}  
  
function getMessageInfo()  
{  
 let val = *randomInt*(-99, 99)  
  
 return new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 *writeLine*(`val=${val}`)  
 if (val > 0)  
 resolve(new MessageInfo(val, "Sözümü tuttum"))  
  
 reject(new MessageInfo(val, "Sözümü tutamadım"))  
 }, 3000)  
 })  
}  
  
function main()  
{  
 let promise = getMessageInfo()  
  
 promise  
 .then(getMessage)  
 .then(message => *writeLine*(message))  
 .catch(messageInfo => *writeLine*(messageInfo.message))  
}  
  
main()

Aşağıdaki örnekte main fonksiyonu içerisinde 2 saniye sonra verinin elde edildiği asenkron bir işlem yapılmıştır:

const *doWork* = timeout => {  
 return new Promise(resolve => {  
 *setTimeout*(() => resolve(Math.random()), timeout)  
 })  
}  
  
function *main*()  
{  
 *doWork*(3000).then(val => console.log(`val=${val}`))  
  
 console.log("main ends")  
}  
  
*main*()

Yukarıdaki program await operatörü ile şu şekilde yazılabilir:

const *doWork* = timeout => {  
 return new Promise(resolve => {  
 *setTimeout*(() => resolve(Math.random()), timeout)  
 })  
}  
  
async function *main*()  
{  
 const val = await *doWork*(2000)  
  
 console.log(`val=${val}`)  
}  
  
*main*()  
  
console.log("here")

Aşağıdaki örneği inceleyiniz:

const *getRandomInt* = (min, max) => *parseInt*(Math.random() \* (max - min) + min)  
  
const *doWork* = timeout => {  
 return new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 let val = *getRandomInt*(-10, 10)  
  
 console.log(`val=${val}`)  
 if (val > 0)  
 resolve(val)  
 else  
 reject("not positive value ")  
 }, timeout)  
 })  
}  
  
function *main*()  
{  
 *doWork*(2000).then(val => console.log(`val=${val}`)).catch(message => console.log(message))  
}  
  
*main*()  
  
console.log("here")

Yukarıdaki örnek await operatörü ile aşağıdaki gibi yapılabilir:

const *getRandomInt* = (min, max) => *parseInt*(Math.random() \* (max - min) + min)  
  
const *doWork* = timeout => {  
 return new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 let val = *getRandomInt*(-10, 10)  
  
 console.log(`val=${val}`)  
 if (val > 0)  
 resolve(val)  
 else  
 reject(new Error("not positive value"))  
 }, timeout)  
 })  
}  
  
async function *main*()  
{  
 try {  
 const val = await *doWork*(2000)  
  
 console.log(`val=${val}`)  
 }  
 catch (err) {  
 console.log(err.message)  
 }  
}  
  
*main*()  
  
console.log("here")