Yeni Başlayanlar için O Programlama Dili

Oytun Özdemir

Ağustos 2020

İçindekiler

Bölüm I

Sistem Tabanlı Programlama Dili

Bölüm 1

Sistem tabanlı programlama dili nedir? (SOFPL)

Bulunduğunuz sistemin altyapısını kullanarak, üzerinde program geliştirmenizi sağlayan programalama dilidir. Hem fonksiyonel hemde sisteme hükmedebileceğiniz bir syntax yapısı vardır. Diğer programlama dili ve sistemsel işlevlerde yazdığınız kodları kısaltma amaçlıdır. Buna bağlı olarak diğer alanlarda bazı işinizi kolaylaştırabilecek özellikler içerir. Aşşağıda verdiğim örnekte C ve O Dili ile yazmış olduğum bir merhaba dünya uygulaması görmektesiniz.

1.1 C ve O Dili için Merhaba Dünya!

```
# include <stdio.h>
int main()
{
    printf("Merhaba Dünya!");
    return 0;
}
```

Printf fonksiyonunu illa bir main içine yazmak zorundasınız. Buna bağlı olarak stdio.h kütühanesini dahil etmek zorunda kaldık. Bunun nedeni printf fonksiyonunun stdio.h kütüphanesinde olmasıdır.

```
show("Merhaba Dünya!")

println("Merhaba Dünya!")

"Merhaba Dünya!"
```

Burada basit olarak show, println bir fonksiyondur. Show sadece göstermek için kullanılır. Println ise satıra metin yada obje bastırmak için kullanılır. Eğer objeyi tek başına yazarsanızda son olarak çıktı verecektir. Herhangi bir kütüphane veya bir şey dahil etmedik çünkü O dilinde yazım (syntax) yapısı nedeni ile zaten hazırda sistem işlevleri gelmekoytunir. Eğerki objenin verdiği sonucu illa görmemiz gerekli ise doğrudan yazarak görebiliriz.

1.2 Nasıl kurulur?

O Dilini kurmak için bir çok metod var. En basiti olang çalıştırılabilir dosyasını indirmek. Buna bağlı olarak docker, go programlama dili ile yazılmış hazır kütüphaneleri bulunmaktadır.

1.2.1 Binary dosyasını indirmek yada edinmek

Bu dosyayı alıp isoytuniğiniz yere taşıdıktan sonra linux'da ***olang dosya adı*** yazarak ola uzantılı dosyalarınızı doğrudan çalıştırabilirsiniz. yada denemek için olang yazıp doğrudan denemeler yapabilirsiniz.

Binary dosyasını bu linkte bulabilirsiniz: `http://gitlab.com/olanguage/olang`

1.2.2 Kaynak koddan derlemek

Eğer hazırda bir go ekosisteminiz varsa şu şekilde kaynaktan en güncel sürümü indirip GOPATH dizininde hem kaynak kodlarına bakabilirsiniz hemde derleyerek kendi testlerinizi yapabilirsiniz. Buna bağlı olarak sizde kendi geliştirmelerinizi yapabilirsiniz.

```
$ go get -v gitlab.com/olanguage/olang
```

Kaynak kodunun bulunduğu adresi altta vereceğim linkten ulaşabilirsiniz.
`http://gitlab.com/olanguage/olang`

1.2.3 Docker üzerinde O dili ile çalışmak

Eğer ben risk almayayım diyorsanız. Docker üzerinde O dili ile geliştirme yapabilirsiniz. Örneğin bir kod yazdınız ama deploy etmek istiyorsanız docker kütüphanesi tam size göredir.

`http://hub.docker.com/repository/docker/olproject/olang`

Docker üzerinde doğrudan kod çalıştırmak için şu şekilde bir yol izleyebilirsiniz.

```
docker run -t olproject/olang olang
```

Alternatif olarak kendi Dockerfile dosyanızı yazarak Docker üzerinde build edebilirsiniz.

```
FROM olproject/olang

RUN ["olang", "dosyaniz.ola"]
```

Bu şekilde docker üzerinde kendi dosyalarınızı ve olang ile yaptığınız çalışmalarıda işletebilirsiniz.

Bölüm 2

Dosyalarla çalışmak

Biraz daha detaylara inecek olursak, o dili sizin ola uzantılı dosyalarınızı işletir yada derler. Bu şekilde birden çok dosyayı dahil ederek işlemlerinize devam edebilirsiniz.

```
def hello = fn(name){
    return ("Merhaba, "+name+"! O dilinden merhaba!")
}
def result = hello("Oytun")
show(result)
```

```
$ olang merhaba.ola

Merhaba Oytun! O dilinden merhaba!
```

Yukarıdaki dosyayı kaydettiniz ve şu şekilde çalıştırdınız. İçindekileri değiştirerek kendinizde pratik yapabilirsiniz. Ben size şimdi ne işlev yaptığımızı tam olarak açıklayayım. Öncelikle basit olarak *hello* fonksiyonu tanımladık. Buna bir *name* değişkeni atadık. En son tüm yaptıklarımızı *result* değişkenine tanımladık ve *show* fonksiyonu ile gösterdik. Bu kadar basit.

2.1 Bilmemiz gereken dosya tipleri

Basit olarak bilmeniz gereken dosya tiplerini tablo olarak belirtiyorum. Nedenlerini diğer bölümlerde kullanarak öğreneceğiz.

Dosya Tipi	Uzantısı	İşlevi
O Language Program	.ola	O Dili Ana Programı yada Program Kodu
O Language Module	.olm	O Dili Modül Dosyası
O Language Process File	Opsfile	Doğrudan bir O dili işlemini çalıştıran dosya (Uzantısı yok)
O Language Pacage File	olpfile.json	O Dili Paket Dosyası
O Language Library File	.oll	O Dili Kütüphane Dosyası

Bölüm 3

Modüllerle çalışmak

Modüller aslında farklı işlevleri dışardan alıp programımıza dahil edip, faklı geliştirmeler yapmamızı sağlar. Kütüphaneler yada ek modüller yapabiliriz.

Şimdi örnek bir module.ola dosyası yazalım.

```
literal hello(name){
    return ("Merhaba, "+name+"! O dilinden merhaba!")
}
```

Modülü yükleyecek örnek bir load.ola dosyası oluşturalım.

```
load "modul.olm"
hello "Oytun";
```

Yaptığımız işlevleri biraz daha detaylandıralım. *modul* dosyamız ve *load* dosyamız var. Biri modül olduğu için "olm" uzantılı olmalıdır. Diğer *load* ise yükleyen ana programımızdır. Ana programımız modülümüzü load eder ve içindeki fonksiyonları ana programımıza aktarır. Burada *literal* literal kavramı bir fonksiyonun literal olduğunu belirtir. Böylece daha önceden *hello* ile ilgili bir şey varsa üzerine yazarak onu değiştirir. Örnekteki gibi *hello* literali oluşturduk. Yani literatürü değiştirdik. Sonrasında bunu çağırdık.

```
Merhaba Oytun! O dilinden merhaba!
```

Sonuç olarak modül ile birlikte ana programımız bize yukarıda sonuçta yer alan çıktıyı verdi.

Bölüm II

Temel Tanımlamalar ve İşlevler

Bölüm 4

Tanımlamalar (Definition)

Bu bölümde daha detaylı işlevlere ve fonksiyonların kullanımlarına yer vereceğiz. Tüm işlevleri açıklayacağız ve nasıl kullanacağınızı anlatacağız.

4.1 Tanımlama nasıl yapılır?

Sabit tanımlama yapmak için kullanılır. Birazdan örnekte göreceğimiz gibi değişken tanımlama yapabileceksiniz. Değişken tanımalamak için *def* kullanılır.

```
def degisken = obje
```

4.2 Türler

Değişken tanılaması yaparken ihtiyacımız olan bazı türler vardır. Bu bölümde tüm türleri tanımlayacağız.

4.2.1 Metin ve Numaralar (String ve Integer)

Metin ve numaralar basit olarak metin veya numara tanımlamanızı sağlar. Metin ve numaraları kullanarak çeşitli matematik algoritmaları yada metin işlemleri yapabilirsiniz. Metin işlemleri " ile başlar ve biter.

```
def string = "bu bir metindir"
```

Numara işlemlerinde ise doğrudan numara kullanmalısınız.

```
def five = 5
```

Gördüğünüz gibi değişken tanımlamak son derece basittir. Bu değişkenleri tüm ana uygulama boyunca kullanabilirsiniz.

4.2.2 Mantıksal İşlemler (Boolean)

Bir şeyin doğru mu yoksa hatalı bir şeymi olup olmadığına bakarken bu tip bir yapı kullanırız. Son derece basittir. Sonuç ya *true* ya *false* döndürülür.

```
def dogru = true

def hatali = false
```

Tanımlamada bu şekilde kullanılır.

4.2.3 Dizi ve Karma Yapılar (Array ve Hash)

Diziler genelde sıralı yapılardır. Bu sıralı yapılar bir liste yapmak gibidir. Listenin her öğesi bir numara alır. Numaralar bu karma yapının bir anahtarı gibidir. Kapıyı açmak için anahtarı kullanarak, kapının arkasındaki öğeye erişmenizi sağlar.

```
def yapilacaklar = [
    "Odanı Topla",
    "Evi Süpür"
]
println(yapilacaklar[0])
```

```
"Odanı Topla"
```

Bu şekilde dizilere bir giriş yapmış olduk. Yapılacaklar listesi oluşturduk ve bu listenin ilk elemanını gösterdik. Karma yapılar üzerine biraz çalışalım. Örneğin anahtarımız ve çeşitli kapılarımız olduğunu düşünelim.

```
def anahtarlik = {
    "anahtar1": "oda 2",
    "anahtar2": "oda 2",
    "anahtar3": "garaj",
    "anahtar4": "alt kapı"
}
println(anahtarlik["anahtar2"])
```

```
"oda 2"
```

Örnekte tüm anahtarları hangi odaları açacağı şeklinde düzenledik. Bu şekilde artık *anahtar2*'nin nereyi açacağını görebiliyoruz. Bu şekilde sizde örneklendirmeler yapabilirsiniz. Yazım şekli oldukça basit. Dizilerde doğrudan "*[]*" kullanarak içine elemanlarınızı yazabilirsiniz. Karma yapılarda ise "*anahtar : değişken*" şeklinde elemanlarınızı tanımlayabilirsiniz. Arasındaki tek fark birinde anahtarınız var diğerinde ise numaralar (indexler) kullanılmaktadır.

4.2.4 Koşullar (If/Else)

Bir koşula bağlı olarak bir işin yapılması veya başka işlerin yapılması gerektiğini belirleyen işlevlerdir. Bu koşullarda *if* (Eğer) ve *else* (Değilse) kullanılır.

```
def hatali = true

if(hatali){
    println("Hataliysam hata yaz 3310'a gönder!")
}else{
    println("Yada sen boşver yazma!")
}
```

```
"Hataliysam hata yaz 3310'a gönder!"
```

Örnekte gördüğünüz gibi *hatali* değişkenimiz *true* olduğu için *if* içindeki satır döndürüldü. Eğer *false* olsaydı *else* kısmındaki koşul çalışacaktır.

4.2.5 Fonksiyonlar (Functions)

Gelelim asıl konumuza, hayatımızın vazgeçilmeden otomatikleştiren fonksiyonlar. **Nedir bu fonksiyonlar?** Fonksiyonlar basit olarak yazdığımız bir işlevi ve değişkenleri çalıştıran işlevlerdir. Çoğu zaman bir şeyi defalarca yapmak isoytuniğimizde tek bir fonksiyon yazarız ve değişkenlerini belirterek defalarca kullanabiliriz.

Örneğin diyelim alışverişe gideceksiniz ve malzeme listeniz yok. Hemen bir örnekle Malzeme listesi işlevleri yazalım.

```
def malzemeler = []

def ekle = fn(malzeme){
    push(malzemeler, malzeme)
    println("Malzeme eklendi!")
}

def goster = fn(num){
    return malzemeler[num]
}

def olmaz = fn(num){
    return ("Senden bi "+(goster(num))+" olmaz!")
}

ekle("pırasa")
ekle("cacık")
ekle("domates")

println(goster(2))
println(olmaz(2))

"cacık"
"Senden bi cacık olmaz!"
```

Burada örneğin fonksiyonları kullandık. Üç adet fonksiyon kullandık. Bu fonksiyonlar *ekle*, *goster*, *olmaz* fonksiyonlarıdır. Hepsi içinde *return* eden şeyi bize geri veriyor yada içinde yazan herşeyi çalıştırarak, sonuçlarını vermekoytunir. Burada *malzemeler* tanımlaması yaptık, içine *ekle* fonksiyonu ile *malzemeler* içine *push* fonksiyonu kullanarak eklenenleri göndermesini sağladık ve *goster* numarasını verdiğimiz malzemeyi bize sonuç olarak döndürdü. Ek olarak sırf eğlence amaçlı *olmaz* fonksiyonunu yazdık. Metinle birleştirerek bize son verilen metni döndürmesini sağladık.

4.2.6 Döngüler (Loops)

Döngüler çoğu zaman hayat kurtarmıştır. Bir listeyi listelemek yada aynı şeyi tekrar yapmamak yerine onu tekrarlayan işlevlerdir. **Döngü yazarken aman diyeyim sonsuz döngü yazmayın.** Sonsuz döngüye giren bir şeyi kimse durduramaz. **Çünkü bu döngüler tehlikelidir.** Sistemi kilitleyebilir. Biz o kadar sonsuza gitmeyen bir döngü yazalım.

```
def i = 0;
loop(i>5){
    println("Boşuna uğraşıyoruz, patronlar bu işten anlamıyor.");
    def i = (i+1);
}
```

```
Boşuna uğraşıyoruz, patronlar bu işten anlamıyor.
Boşuna uğraşıyoruz, patronlar bu işten anlamıyor.
Boşuna uğraşıyoruz, patronlar bu işten anlamıyor.
Boşuna uğraşıyoruz, patronlar bu işten anlamıyor.
Boşuna uğraşıyoruz, patronlar bu işten anlamıyor.
Boşuna uğraşıyoruz, patronlar bu işten anlamıyor.
```

Burada basit olarak *i* değişkeni tanımladık. *loop* kullanarak, *i* 5'den büyük olana kadar çalıştırdık ve her çalışmada *i* sayısına bir sayı daha ekledik. Bu *loop* fonksiyonun çalışma şeklidir. *loop* içinde belirtilenler eşitlenene kadar çalışmaya devam eder.

Birde bu döngüyü daha farklı olarak inceleyelim. Biraz daha kısıtlama getirelim. Döngüde daha önceden yazmış olduğum *yapilacaklar* kısmını kullanacağım. Bu listedeki öğeleri görmek ve içeriklerine erişerek bir liste yapalım.

```
def yapilacaklar = [
    "Odanı Topla",
    "Evi Süpür",
    "Yemek Pişir",
    "Ödevlerini Yap"
]

for(yapilacaklar in numara,oge){
    def metin = (itostr(numara+1)+". "+(oge));
    println(metin)
}
```

```
1. Odanı Topla
2. Evi Süpür
3. Yemek Pişir
4. Ödevlerini Yap
```

Örnekte gördüğünüz gibi tüm yapılacak listesini düzenledik. Öncelikle burada *yapilacaklar* listemiz vardı. Bu listeyi *for* döngüsünde her elemanın *numara* ve *oge* tanımlanacak şekilde ayarladık ve *metin* degiskeninde düzenleyip, çıktısını aldık. Kullandığımız tek farklı şey numarayı *itostr* fonksiyonu ile string'e çevirdik. Bunu yapmamızın nedeni ise **numara ve metin toplanamaz.**

4.2.7 Literatür Yapıları (Literals)

Litaretür yapısı sistemdeki var olan syntax üzerinde değişiklik yapabilmemizi sağlar. Literallerin fonksiyonlardan farkı ise programlama diline bir fonksiyon ekleme anlamı taşır. Sistemdeki fonksiyonları değiştirebilmeniz yada müdahale edebiliyor olmanız, size dilde yazılmış fonksiyon yada metodları genişletebilmenizi sağlar. Genellikle modüllerde kullanılır.

Şimdi size örnek olarak bir matematik kütüphanesi yazdıracağım. Bu dosyanın ismi *math.olm* olacak. Bu kütüphanenin amacı sadece basit hesap makinası işlevlerini modül haline getireceğiz.

```
literal topla(x,y){
    return (x+y)
}

literal cikar(x,y){
    return (x-y)
}

literal carp(x,y){
    return (x*y)
}

literal bol(x,y){
    if(y == 0){
        return "sonsuz"
    }
    return (x/y)
}

literal fak(x){
    def i = 0;
    def sonuc = 1;
    loop(i==x){
        def sonuc = (sonuc * x);
        def i = (i+1);
    }
    return sonuc;
}
```

Matematik kütüphanemiz işlevlerini yapmaya hazır. Hazırladığımız litaretürü kullanarak örnek bir program yazalım. Program *fak* fonksiyonunda faktöriyel işlemi çalıştırsın. Ek olarak *topla* fonksiyonunu çalıştıralım.

```
load "math.olm"

println(fak(12))
println(topla(1,2))

8916100448256
3
```

Gördüğünüz gibi sonuç olarak *fak(12)* ve *topla(1,2)* sonuçlarını elde ettik. Sizde bu örnek kütüphaneyi baz alarak kendi literatürünüzü geliştirebilirsiniz.

4.2.8 Kapsam Yapısı (Scopes)

Kapsam yapısı çoğu zaman sınıf yapılarına benzetilmiştir. Burada kapsam yapısı bir çok şeyi tek bir kapsamda toplamak ve bütünleştirmek amacı ile kullanılır. Az önce yazdığımız *math.olm* modülünü kapsama alalım ve modülün bir kapsam içinde bulunsun.

```
scope math{
    def topla = fn(x,y){
        return (x+y)
    }

    def cikar = fn(x,y){
        return (x-y)
    }

    def carp = fn(x,y){
        return (x*y)
    }

    def bol = fn(x,y){
        if(y == 0){
            return "sonsuz"
        }
        return (x/y)
    }

    def fak = fn(x){
        def i = 0;
        def sonuc = 1;
        loop(i==x){
            def sonuc = (sonuc * x);
            def i = (i+1);
        }
        return sonuc;
    }
}
```

Örnekte bir matematik kapsamı oluşturduk. İçine fonksiyonlar ekledik. Bu *math* kapsamının içinde fonksiyonlar grubu oluşturulur ve içine fonksiyonlar atanır.

```
load "scope.olm"

def sonuc = math->fak(10)
println(sonuc)
```

```
10000000000
```

Modülümüzü çağırdık ve *math* kapsamı otomatik olarak geldi. *-¿* parametresi ile kapsamın içinde yer alan *fak* fonksiyonuna bir parametre gönderdik (*fak* fonksiyonuna erişmiş olduk). Sonucun çıktısını aldık.

4.2.9 Hata ayıklama (Exceptions)

Her programlama dilinde bir hata ayıklama bölümü vardır. Biz buna kısaca "Ayıkla princin taşını!" deriz. Program önce bir süreç başlatır ve süreçte bir şey hatalı giderse başka bir şey dönecektir. Eğer hata olmazsa sonuç çıktısı verir. Bu süreç yönetiminde olan hataları ayıklamamızı sağlar. Nasıl kullanıldığına bir göz atalım. Burada bilmeniz gereken tipler; *begin*, *except*, *recover* ve *final* türleridir. Bunların detaylarını bir örnekle açıklayalım.

```
begin hata {
    def hatali = true

    if(hatali){
        except hata "Bozdun!"
    }else{
        final hata{
            println("Hata mı ne hatası!")
        }
    }

    recover hata {
        println(error)
    }
}
```

```
Bozdun!
```

Bir *hata* tanımlaması başlattık ve bu tanımlamaya *hatali* değişkeni tanımladık. Burada değişkeni içerde veya dışarda tanımlamanızın bir önemi yoktur. Eğer *hatali* değişkeni *true* (doğru) ise *hata* tanımlaması bir hata süreci oluşturacaktır. *recover* (kurtarma) metodunu *hata* tanımlamasına uygulayacaktır. Buna bağlı olarak *except* üzerinden gelen mesajı *error* değişkenine tanımlar ve isoytuniğiniz şekilde kullanabilirsiniz. Eğer süreçte bir sorun oluşmazsa *hata* sürecine *final* gönderilerek sürecin sorunsuz tamamlanması sağlanır.

Bölüm 5

Girdi ve Çıktı İşlemleri (Input/Output)

Terminalden kullanıcıdan bilgi almak yada ekrana bir şey yazdırmak için bir kaç metod var. Yeni versiyona göre O dili üzerinde *input* ve *output* fonksiyonları *show* ve *println* işlevlerine bağlı olarak farklı iki ekstra opsiyon daha sağlamaktadır.

```
//input fonksiyonu:
def veri = (input("Veri giriniz: "))

//output fonksiyonu:
output(veri+" girdiniz.")

Veri giriniz: veri
veri girdiniz.
```

Örnekte görüldüğü gibi *input* terminalden bir veri girişi aldık ve bunu *veri* değişkenine eşitledik. Sonra *output* fonksiyonu ile çıktısını aldık. Bu işlevler teknik olarak *show* ve *println* fonksiyonların benzeridir.

Bölüm 6

Matematik Operatörleri

Matematik operatörleri hayatımızın olmazsa olmazlarından biri oldu. Bu operatörlerle temel dört işlem yapmamak dışında fonksiyonlarımızda bu operatörleri kullanır hale geldik. Bu işlevler O Dilin'de son derecek basittir.

6.1 Temel İşlevler

Bu bölümde basit hesaplama işlevlerini göreceğiz. Bunları nasıl kullanacağımızı ve tanımlamalarda kullanacağımızı öğreneceğiz.

6.1.1 Toplama

Toplama işlemi için artı + işareti kullanılır.

```
def sonuc = 2 + 2
inspect(sonuc)
```

```
4
```

6.1.2 Çıkarma

Çıkarma işlemi için eksi - işareti kullanılır.

```
def sonuc = 2 - 2
inspect(sonuc)
```

```
0
```

6.1.3 Bölme

Bölme işlemi için eğik çizgi / işareti kullanılır.

```
def sonuc = 2 / 2
inspect(sonuc)
```

```
1
```

6.1.4 Çarpma

Çarpma işlemi için yıldız * işareti kullanılır.

```
def sonuc = 2 * 2
inspect(sonuc)
```

```
4
```

Şimdilik bu işlemlerle matematik işlemleri yapabilirsiniz.

6.2 Yardımcı İşlevler

Temel matematik operatörleri için işinize yarayabilecek bir kaç basit fonksiyon O dili ile gelmektedir. Bunları ihtiyaç duyduğunuzda kullanabilirsiniz.

6.2.1 Toplam (Sum)

Toplam işlevi, matematikteki toplam işlevine benzerdir. Fakat burada dikkat edilmesi gereken iki elemanı ilk iki değişkende toplayarak bunu son değişkende belirlenen küme yada fonksiyona uygular.

```
def sonuc = (math_sum(1,2, [1,2,3,4]))
inspect(sonuc)
```

```
[3, 9, 18, 30]
```

Gördüğünüz gibi doğrudan belirlenmiş ikinci dizideki her elemana 1 ve 2 rakamlarını toplayıp, kümeye uyguladı. Bunu daha sonra sizlere Haritalama İşlevleri bölümünde daha detaylı anlatacağız.

6.2.2 Mutlak Değer (Abs)

Mutlak değerini almak istediğiniz rakamı vermeniz durumunda sonuç olarak size bir sayının mutlak değerini verecektir. Burada *abs* fonksiyonu kullanılır.

```
math_abs(-12)
```

```
12
```

Basit olarak mutlak değerini alıp sonuç döndürecektir.

Bölüm 7

Geometri Fonksiyonları

Geometrik işlemler yapmanız için bazı basit işlevler O Dili üzernde gelmektedir. Bu Yardımcı fonksiyonları sizlerle detaylandıralım.

7.0.1 Logaritma (Log)

Logaritma almak için *log* fonksiyonu kullanabilirsiniz.

```
math_log(12)
```

```
2.4849066497880004
```

7.0.2 Kosinüs (Cos)

Kosinüs almak için *cos* fonksiyonunu kullanabilirsiniz.

```
math_cos(12)
```

```
0.8438539587324921
```

7.0.3 Tanjant (Tan)

Tanjant almak için *tan* fonksiyonunu kullanabilirsiniz.

```
math_tan(12)
```

```
-0.6358599286615807
```

Bölüm 8

Karşılaştırma İşlevleri

Bu bölümde iki ifadeyi karşılaştırmak için kullanacağınız operatörleri vereceğiz. Bu operatörlerle karşılaştırma yapabilirsiniz. Ek olarak yaptığınız karşılaştırmaları iç fonksiyon ve değişkenlerde kullanabilirsiniz.

8.1 Eşittir

İki öğeyi karşılaştırıp iki öğrenin eşit olduğunda doğru sonucu almanızı sağlayan eşittir operatörü iki öğeyi karşılaştırıp doğru veya yalnış sonuç verecektir.

```
def hatali = (1 == 2)
def dogru = (1 == 1)

inspect(hatali)
inspect(dogru)
```

```
false
true
```

8.2 Eşit Değildir

İki öğeyi karşılaştırıp iki öğrenin eşit değilse doğru sonucu almanızı sağlayan eşittir operatörü iki öğeyi karşılaştırıp doğru veya yalnış sonuç verecektir.

```
def hatali = (1 != 2)
def dogru = (1 != 1)

inspect(hatali)
inspect(dogru)
```

```
true
false
```

8.3 Büyüktür

Bir elemanın diğerinden büyük olup olmadığına bakarak doğru veya hatalı sonuç sağlayan operatördür.

```
def kardes = 10
def abi = 20

inspect(kardes > abi)
inspect(abi > kardes)
```

```
false
true
```

8.4 Küçüktür

Bir elemanın diğerinden küçük olup olmadığına bakarak doğru veya hatalı sonuç sağlayan operatördür.

```
def kardes = 10
def abi = 20

inspect(kardes < abi)
inspect(abi < kardes)
```

```
true
false
```

8.5 Büyük ve Eşittir

Bir eleman diğerine büyük ve eşitmi kontrol edip sonuç olarak doğru veya yalnış sonuç döndüren operatördür.

```
def kardes = 10
def abla = 20
def abi = 20

inspect(abla >= abi)
inspect(kardes >= abla)
inspect(kardes >= abi)
```

```
true
false
false
```

8.6 Küçük ve Eşittir

Bir eleman diğerine küçük ve eşitmi kontrol edip sonuç olarak doğru veya yalnış sonuç döndüren operatördür.

```
def kardes = 10
def abla = 20
def abi = 20

inspect(abla <= abi)
inspect(kardes <= abla)
inspect(kardes <= abi)
```

```
true
true
true
```

Bölüm 9

Haritalama İşlevleri (Mapping)

Haritalama işlevleri bir programlama dilinde bir elemandaki her elemanı kümeleyerek, fonksiyona uygulama işlevidir. Kümenin her elemanı fonksiyonda bir değişkene tanımlanır. Dizi ise tek eleman kullanılır. Karma yapılarda ise iki eleman kullanılır.

```
def kume = [1,10, 30, 40, 50]

def sonuc = (map(fn(x){
    return x*x
}, kume))

inspect(sonuc)

[1, 100, 900, 1600, 2500]
```

Gördüğünüz gibi dizideki tüm elemanları kendisi ile çarptık ve diziye geri döndürdük.

Bölüm 10

Dosya İşlemleri

Dosyalarla çalışmak son derece kolaydır. Okuma ve yazma işlevleri ile başlayalım. Dosyalar üzerine işlem yapmak her zaman bir dosyayı değiştirmek yada içeriği üzerinde çalışmak yapmak her ortamda en gerekli olan özelliktir. Bazı programlama dilleri bu işleri olabildiğince uzun tutmaktadır. Burada göreceğiniz fonksiyonlar doğrudan sistemle alakalıdır.

10.1 Dosya Okuma (File Read)

Dosya okumak için basit olarak *read* kullanırız. Dosyayı okutup bir değişkene atamamız yeterlidir.

```
def deneme = read("deneme.txt")
```

Dosyamızı *read* fonksiyonu ile okuttuk. Son olarak deneme değişkenine eşitledik. Sonuç metin dönecektir.

10.2 Dosya Yazma (File Write)

Dosya yazmak için basit olarak *write* kullanılır. Herhangi bir stringi yada bir sonucu doğrudan dosyaya yazdırır.

```
def icerik = "Bu bir denemedir";
write("deneme.txt", icerik)
```

Örnekteki gibi *icerik* metnimizi *deneme.txt* dosyamıza yazdırdık.

10.3 Dosya Silme (File Delete)

Dosya silmek için basit olarak *remove* kullanılır. Doğrudan dosyayı sistem tabanında siler.

```
remove("deneme.txt")
```

Örnekte *deneme.txt* dosyamızı sildik.

Bölüm 11

Dizin İşlemleri

Dizinlerle ilgili basit işlemler yapmak için bir kaç fonksiyon öğreneceksiniz. Bu bölümde dizinler üzerinde çalışmalar yapabilirsiniz. Öncelikle dizin oluşturma işlevinden bahsedelim. Sizin varolan kullanıcınız ile dizin oluşturmanızı sağlar.

```
mkdir("dosyam_benim")
```

mkdir fonksiyonu ile dizinimizi oluşturduk. Şimdi bu dizini silelim.

```
rmdir("dosyam_benim")
```

Dizin ve içinde ne varsa silindi. Şimdi bu dosya ve dizin oluşturma işlevleri ile bir program yazalım.

```
scope sys{
    def oku = fn(dosya){
        return read(dosya)
    }

    def ucur = fn(dosya){
        return remove(dosya)
    }

    def yaz = fn(dosya, icerik){
        return write(dosya, icerik)
    }

    def milletGorsun = fn(dosya){
        return chmod(dosya, 777)
    }

    def benGoreyim = fn(dosya){
        return chmod(dosya, 755)
    }
}
```

Şu fonksiyonları sys kapsamına oluşturduk; *oku*, *yaz*, *ucur*, *milletGorsun* ve *benGoreyim*. Bu fonksiyonlar isminde yazan işlevleri gerçekleştirmekoytunir.

Sadece iki adet fonksiyon *chmod* (yetki değiştirme) fonksiyonunu çağırır. Herkesin görmesi için *777* yetkisi verilir. Sadece şu an kullanıcının görebilmesi için ise *755* yetkisi verilir. Bu kısımların detaylarına *Yetkiler* bölümünde daha detaylı değineceğiz.

```
load "sys.olm"

sys->yaz("deneme.txt", "deneme")
def deneme = sys->oku("deneme.txt")
sys->milletGorsun("deneme.txt")
sys->ucur("deneme.txt")
```

Modülümüzü çağırdık ve *deneme.txt* dosyamıza metin yazdırdık ve okuduk. Herkesin görmesi gerektiğini belirten fonksiyonu çağırdık. Sonra dosyamızı sildik. Hepsi *sys* kapsamında yer alan fonksiyonlardır.

Bölüm 12

Sistemsel İşlevler

İşletim sistemi tarafını ilgilendiren değişikliklere değineceğiz. Bu bölüm sadece işletim sistemi üzerinde yapabileceğiniz bazı önemli düzenlemeler yada işlevlerden oluşacaktır.

12.1 Yetkiler (File Permissions)

Eğer linux/unix bilginiz varsa yetkilere hakim olduğunuzu varsayıyoruz. Eğer hakım değilseniz bu bölümde yetkilerden bahsedeceğiz. Sistem tarafında yetkilendirme hangi dosyaya yada hangi klasöre erişebileceğinizi kısıtlayan yada arttıran işlevlerin bütünü yetkilendirmedir. Örneğin bir dosya oluşturdunuz okunmasını istemiyorsunuz. Buna gerekli yetkiyi okuduktan sonra uygularsanız, kullanıcı erişemeyecektir. Sadece program tarafından dosya okunacaktır. Gerekli işlemler okunduktan sonra yapılacak şekilde programlayabiliriz.

```
def dosyam = "deneme.txt";
def oku = read(dosyam)
chmod(dosyam, 000)
println(oku)
chmod(dosyam, 755)
```

Örnekte gördüğünüz gibi dosyayı okudum ve *000* (oct) yetkilendirmesini uyguladım. Dosya program tarafında okunabiliyor ama kullanıcı açmaya çalıştığında yetki hatası alacaktır. Sonrasında tekrar yetkiyi geri verdim. Bu program tarafında herhangi bir dosya okunurken kilitleyebilirsiniz. Bunun için *chmod* fonksiyonu yeterlidir. Birazdan bu yetkileri nasıl hesaplanacağını anlatacağım.

Parametreler ve anlamlarını aşşağıdaki tabloda bulabilirsiniz. Hesaplama yapmak için ise bu yetkileri toplayın. Ve yan yana üçlü şekilde yazın.

Örneğin bir yazma gruba ve kullanıcıya yazma izni vermek için;

```
4+2+0 = 6
4+2+0 = 6
0+0+0 = 0

chmod(dosya, 660)
```

Şeklinde kullanabilirsiniz.

Parametre	Anlamı
u	Kullanıcı
g	Grup
o	Diğer
a	Herkes
r	Oku
w	Yaz
+	İzin Ver
-	İzin Al
4	Okuma İzni (Numeric)
2	Yazma İzni (Numeric)
1	Çalıştırma İzni (Numeric)

12.2 Sistem Değişkenleri (Environment)

Sistem değişkenleri sistemin değişkenlerine erişebilmenizi sağlar. Ek olarak bu değişkenler üzerinde işlem yapabilirsiniz.

```
def path = sysenv("PATH")
```

Örneğin burada sistem'deki PATH değişkenini kontrol ettik ve biz kendi *path* değişkenimize eşitledik.

12.3 Sistem Komutları ve İşlemler (Process)

Sistemde başka komutlar çalıştırmak ve bunları işlevlendirmek için kullanmanız gereken fonksiyonları bu bölümde özetleyeceğiz.

Sistemde bu tarz farklı işlemleri başlatabilmeniz için *proc* kullanılır. Örneğin bir dizindeki dosyaları farklı bir şekilde listeletmek istiyorsak linux/unix platformlarda *ls* komutunu kulanırız.

```
def ls = (proc ls "ls" "-l")
println(ls)

.
..
forloop.ola
hello.c
hello.ola
load.ola
loop.ola
math.olm
mathtest.ola
merhaba.ola
modul.olm
proc.olm
scope.olm
scopetest.ola
sock.olm
sys.olm
systest.ola
yapilacaklar.ola
```

İşlemin sonucunu isterseniz bir değişkene tanımlayabilirsiniz.

12.4 Bağlantılar (Connections)

Sistem üzerinden bağlantı yapmak için genelde port kullanırız. Port açtıktan sonra başka bir sistemden port'a bağlantı kurarak iki bilgisayarı haberleştirebiliriz. Birazdan bu ikili bağlantıları nasıl yapacağımızı göreceğiz.

12.4.1 Soket Açma (Socket)

Bağlantı kurabilmek için öncelikle bir port açmak gereklidir. Bu işlemi yapabilmek için *sock* yapısını kullanmalıyız.

```
sock socket "tcp4" "9050" "0.0.0.0";
def messages = {
    "ping": "pong"
}
sock_listen(socket, messages);
```

Örnekte görüldüğü gibi *socket* değişkenine bir socket açtık. *tcp4 9050* portundan gelen mesajlara cevap verecek şekilde portu dinlemeye aldık. Karşı taraftan *ping* mesajı gelirse *pong* cevabı verilecektir.

12.4.2 Socket'e Bağlanma (Socket Connection)

Bir porta bağlanabilmek için öncelikle socket'i açmanız gereklidir. Sonrasında send ile mesajlarını port'a raw olarak gönderebilirsiniz.

```
sock socket "tcp4" "9050" "0.0.0.0"
sock_send(socket, "ping")

"pong"
```

Örnekteki gibi socket açtıktan sonra *send* ile mesajımızı gödnerdik. Sadece dikkat etmeniz gereken bağlantı yapan ve bağlantıya erişen programların iki ayrı program olmasıdır.

```
client: send -> ping -> listen (9050)
server: listen (9050) -> pong -> send
```

Yukarıda belirtilen şekilde örneklendirilebilir.

Bölüm III

Web ve Veritabanı Yönetimi

Bölüm 13

Web Sunucusu

Bu bölümde web tabanlı çalışmalarla sistem içinde yer alan yardımcı fonksiyonlarla bir web sitesi çalıştıracağız. Daha Sonra Yaptığımız web sitesini basit olarak veri tabanına bağlayacağız. Yönlendirme ve veri tabanları ile ilgili genel çalışmaların O dili üzerinde nasıl yazılabileceğini öğreneceksiniz.

13.1 Yönlendiriciler (Routers)

Yönlendirici (Router), herhangi bir metod ile 80 veya 443 portlarına yapılan sorgulardan oluşan yapılardır. Bu yönlendiricilere sorgulama yapabilmek için bazı başlık bilgileri gönderilir ve bu başlık bilgilerine karşı taraftaki makinanın yanıt vermesi istenir. Eğer böyle bir şey yok ise çeşitli hata kodları döndürülür. O Dilinde bu yönlendiricileri yazmak için *webserver* fonksiyonu kullanılır. Bu fonksiyona çeşitli ayarlamalar verilerek otomatik olarak bir web tabanlı yönlendirici oluşturulur. Kod çalıştırıldığında ise sonuç olarak bir web server çalışır.

```
def config = {
  "yonlendirme adresi": {
    "type": "kullanılacak metod :
                |GET
                |POST
                |PUT
                |DELETE
                |PATCH
                |OPTIONS
                |HOST
                |UPLOAD
                |STATIC
                |STREAM",
    "response": "cevap içeriği",
    "input": "giriş isteniyorsa eğer input bilgileri",
    "maxsize": "dosya yüklenecekse limiti (MB olarak)",
    "headers": "başlık bilgileri",
    "folder": "klasör gerekli ise klasör",
    "data": "eğer veri gönderilecek ise",
    "host": "hangi bilgisayardan bu veriler kabul edilecek?",
```

```
  }
}

webserver(port, config);
```

Yukarıda belirtildiği şekilde bir ayar karma dizisi ve port belirterek web sunucusu oluşturabilirsiniz. Kullanabilmeniz için tüm detayları belirttik. Metodların kullanım şekillerine bir bakalım.

13.1.1 GET

Get metodu sadece veri çekmek için kullanılan metodtur.

```
"/get": {
  type: "GET",
  response: "Cevap Mesajı"
  data: {}
}
```

Kullanım şekli yukarıdaki gibidir. Diğer girilen değerler geçersizdir.

13.1.2 POST

Post metodu sunucuya veri gönderilebilmesini sağlar. Ek olarak data kısmında parametreler belirtilebilir.

```
"/post": {
    type: "POST",
    response: "Merhaba {{.parametre}}!"
    data: {
        parametre: "Oytun"
    }
}
```

Parametreler şu şekilde cevap içine render edilebilir;

```
{{.parametreadi}}
```

Ek olarak data kısmında hangi parametrelerin render edileceği verilmelidir.

13.1.3 PUT

Put metodu içine atma metodudur. Post ile benzer çalışır. HTTP/2 ile birlikte gelen bir özelliktir.

```
"/put": {
  type: "PUT",
  response: "Merhaba {{.name}}!"
  data: {
    name: "Oytun"
  }
}
```

```
{{.parametreadi}}
```

Ek olarak data kısmında hangi parametrelerin render edileceği verilmelidir.

13.1.4 UPLOAD

Upload yükleme işlevlerini gerçekleştirir. Post metodu kullanılarak bu adrese dosyalar formdan yada herhangi bir şekilde gönderildiğinde otomatik olarak *folder* parametresinde belirtilen klasöre kaydedilir. Ek olarak formdan gelen *input* kabul edilir. Dosya yükleme limiti belirtilecek ise *maxsize* MB olarak belirtilir. Bu belirtilmediği taktirde hata alabilirsiniz. O yüzden programınızı yazarken dosya yükleme limitini işinize yarayabilecek en yüksek değerde tutabilirsiniz.

```
"/upload": {
  type: "UPLOAD",
  input: "files",
  maxsize: "512",
  folder: "upload/"
}
```

13.1.5 DELETE

Delete HTTP/2 ile birlikte gelen silme metodudur. Bu metodta silme işlevleri yapılabilir.

```
"/delete": {
  type: "DELETE",
  response: "Sakın Silme! {{.name}}!"
  data: {
    name: ""
  }
}
```

13.1.6 STATIC

Static, sabit dosyaları listelemek ve geri döndürme amaçlı kullanılan bir metodtur. Sadece klasörde bulunan dosyaları listeler.

```
"/public/*filepath":{
    "type": "STATIC",
    "responsе": "/public/*filepath"
}
```

Burada dikkat edilmesi gereken sadece *filepath* adresidir. Eşleştirilip ilgili dosya geri çağrılır.

Şimdi bir örnek ile gerçek bir web sunucusu oluşturalım.

```
def config = {
    "/": {
        "type": "GET",
        "response": "O Dili yönlendiricisine hoş geldiniz!"
    }
}
webserver("8080",  config);
```

```
$ curl localhost:8080
O Dili yönlendiricisine hoş geldiniz!
```

Örnekteki 8080 portuna bir web sunucusu açtık ve anasayfadan GET metodu ile gelen sorgularda cevap vermesini sağladık. Sizde bu şekilde ayarları ve yönlendirmeleri kullanarak kendi web sunucularınızı oluşturabilirsiniz.

Bölüm 14

Veritabanı Yönetimi (Database)

Bu bölümde veri tabanı ile O dili üzerinde nasıl çalışma yapılır. Basit olarak giriş yapacağız. Modeller oluşturacağız ve bağlantılar kurarak veri tabanı üzerinde çalışma yapacağız. Veri tabanına bağlanmak ve veri çekmek için veri tabanları kullanılır. Veri tabanları genelde çok büyük yada küçük verileri tutmak için kullanılabilir. O dili kendi içinde veri tabanı yapısı içermekoytunir. İsterseniz dış veri tabanları ile bağlantı kurabilirsiniz. Yada iç veri tabanında dosya tabanlı çalışmalar yapabilirsiniz.

14.1 Modeller ve Bağlantılar (Model and Connections)

Veri tabanını kullanabilmek için öncelikle bağlantı kurmanız gerekli. Nasıl bağlantı kurabileceğiniz ile ilgili örnek kodlarla birlikte basit bir Model çalışması yapacağız.

```
def db = (database("tipi","bağlantı adresi"))
```

Database fonksiyonu kullanılarak, veri tabanı ve adresini yazarak bağlantıyı *db* değişkenine bağladık.

```
def model = (model(db, "tablo", {
    "id": "int"
    ....
}))
```

Model fonksiyonunu kullanarak, bağlantı kurduğumuz veri tabanına bir tabloyu modelimize bağladık.

Şimdi hepsini basit bir örnekte toplayalım ve detaylı olarak açıklayalım.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
```

```
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
}))
```

Bir *baglanti* tanımlaması yaptık ve veri tabanımızı *internal* olarak belirledik. İç veri tabanımızı kullanıyoruz. Bu veri tabanı *veritabani* klasöründe depolanacak şekilde ayarladık. Bir kullanıcı modeli oluşturduk ve bunu *kullanici* modeli olarak ilerleyen bölümlerde kullanacağız.

14.2 Tablo Oluşturma ve Silme (Migrate/Drop/Truncate)

Eğer veritabanı *migrate* ve *drop* işlemlerini destekliyorsa bu bölümde tablo silme ve oluşturma işlevlerinden bahsedeceğiz.

```
//migrate fonksiyonu:
migrate(model)
//drop fonksiyonu:
drop(model)
```

migrate fonksiyonu daha önceden modelini yazdığımız bir objeyi otomatik olarak veri tabanına oluşturur ve bunun geçmişini kaydeder. *drop* ise bunun tam tersidir. Belirtilen veri tabanını imha eder. Bunu bir örnekle detaylandıralım.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
}))

//drop fonksiyonu:
drop(kullanici)

//migrate fonksiyonu:
migrate(kullanici)
```

Veri tabanımızda bir kullanici modeli oluşturduk ve model varsa önce *drop* fonksiyonu ile sildik. Sonrasında *migrate* fonksiyonu ile tablomuzu yeniden oluşturduk.

Bu durumlara alternatif olarak *truncate* fonksiyonuda kullanılabilir. Bu fonksiyon tabloyu silmek yerine Tablonun içindekileri silmekoytunir.

```
truncate(model)
```

Modelin içinde yer alan tüm veriler silinecektir.

14.3 Oluşturma (Create)

Veri tabanında veri oluştturmak için kullanacağımız *create* fonksiyonumuz mevcut. Bu fonksiyon ile oluşturduğumuz modellere veri kaydetmeye başlayabiliriz. Basit olarak kullanım şeklini aşşağıdaki gibidir.

```
create(model, {...veriler...})
```

Veriler kısmında veriler yer alacağı için bunu bir örnekle gösterelim.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
}))

// create fonksiyonu:
create(kullanici, {
    "id": "1",
    "isim": "Oytun",
    "soyisim": "Ozdemir",
    "kullanici": "oytun",
    "sifre": "1234"
})
```

Daha önceden kurduğumuz bağlantı ve modelimiz üzerine veri oluşturduk. *kullanici* modelimize data göndererek verilerimizi oluşturduk.

14.4 Güncelleme (Update)

Veri tabanımızda verileri güncelleme yapmak için *update* kullanıyoruz. Daha önceden yaptığımız bir hatayı düzeltmek yada veri güncellemek için kullanabiliriz. Basit olarak kullanım şekli aşşağıdaki gibidir.

```
update(model, {...veriler...})
```

Veriler kısmından göndereceğiniz veriler eskisi ile değiştirilir. Yerine yeni veriler yazılır. Bunu bir örnekle detaylandıralım.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
```

```
}))

// create fonksiyonu:
create(kullanici, {
    "id": "1",
    "isim": "Oytun",
    "soyisim": "Ozdemir",
    "kullanici": "oytun",
    "sifre": "1234"
})

//update fonksiyonu:
update(kullanici, {
    "id": 1,
},{
    "sifre": "gaiB3woo"
})
```

Örnekte en alt kısımda *update* fonksiyonunu kullanarak şifremi güncelledim. Sizde verilerinizi güncelleyerek deneme yapabilirsiniz.

14.5 Veri Çekme (Fetch)

Veri çekmek ve veri tabanından belirli bir veriye göre alakalı verileri bir değişkene atamak için *fetch* metodumuz mevcut. Bu metodu kullanarak veri çekebilirsiniz.

```
def sonuc = (fetch(model, {...sorgulanacak veri...}))
```

Sorgulanacak veri kısmına neye ihtiyacınız varsa belirterek, bununla alakalı olan verileri çekip bir diziye eşitleyebilirsiniz. Şimdi bunu bir örnekle detaylandıralım.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
}))

// create fonksiyonu:
create(kullanici, {
    "id": "1",
    "isim": "Oytun",
    "soyisim": "Ozdemir",
    "kullanici": "oytun",
    "sifre": "1234"
})
```

```
//update fonksiyonu:
update(kullanici, {
    "id": 1,
},{
    "sifre": "gaiB3woo"
})

//fetch fonksiyonu:
def kim = (fetch(kullanici, {"id": 1})
inspect(kim)

{
    "id": "1",
    "isim": "Oytun",
    "soyisim": "Ozdemir",
    "kullanici": "oytun",
    "sifre": "1234"
}
```

Örnekte gördüğünüz gibi ID'si 1 olan kullanıcıyı çekip *kim* değişkenine eşitledim ve *inspect* ile gelen veriye baktık.

14.6 Sorgu Yazma (Query)

Eğer veri tabanınız sorgu çalıştırmanıza izin veriyorsa sorgu çalıştırabilirsiniz.

```
def sonuc = (query(model, "sorgunuz", {...parametreler..}))
```

Sorgu çalıştırabilmek için SQL yada benzeri bir veri tabanı bilmeniz gerekmekoytunir. SQL bilginiz varsa doğrudan sorgular çalıştırabilmenize olanak tanır. Bunu bir örnekle deneyelim.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
}))

// create fonksiyonu:
create(kullanici, {
    "id": "1",
    "isim": "Oytun",
    "soyisim": "Ozdemir",
    "kullanici": "oytun",
    "sifre": "1234"
```

```
})

//update fonksiyonu:
update(kullanici, {
    "id": 1,
},{
    "sifre": "gaiB3woo"
})

//fetch fonksiyonu:
def kim = (fetch(kullanici, {"id": 1})
inspect(kim)

//query fonksiyonu:
def kullanicilar = (query(user,
        "SELECT * FROM kullanicilar LIMIT ?",
        100))
```

Son yazdığım *query* fonksiyonu iç veri tabanım desteklemediği için çalışmayacaktır. Fakat desteklenen veri tabanlarında SQL sorgusunu çalıştırır ve *kullanicilar* değişkenimize gönderir. Bu sorgunun amacı 100 adet kullanıcıyı limitleyerek çekmektir. SQL bilginiz varsa bu şekilde sorgular yazabilirsiniz. Burada dikkat etmeniz gereken önemli bir nokta; SQL sorgusuna giden parametreler filtrelenerek sorguya eklenir ve SQL Injection sorunu yaşamazsınız. SQL Injection bir güvenlik problemidir. Parametreye eklenilen sorgularında çalışmasını sağlayan bir güvenlik sorunudur. O dilinde bu parametreler otomatik olarak denetlenerek parametre olarak girilen şeylerin otomatik olarak temizlenmesi sağlanır.

14.7 Silme (Delete)

Silme işlemleri veri tabanından sorgu yada parametreler kullanılarak verilerin silinmesini sağlar. Burada *delete* fonksiyonu kullanılır.

```
delete(model, {...parametreler...})
```

Model kullanılarak parametrelere bağlı veriler silinecektir. Bunu bir örnekle pekiştirelim.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
}))

// create fonksiyonu:
create(kullanici, {
```

```
    "id": "1",
    "isim": "Oytun",
    "soyisim": "Ozdemir",
    "kullanici": "oytun",
    "sifre": "1234"
})

//fetch fonksiyonu:
def kim = (fetch(kullanici, {"id": 1}))
inspect(kim)

//query fonksiyonu:
def kullanicilar = (query(user,
        "SELECT * FROM kullanicilar LIMIT ?",
        100))

//delete fonksiyonu:
delete(kullanici, {"id": 1})
```

Örnekte görüldüğü gibi ID'si 1 olan kullanıcıyı sildik.

Veri tabanı ile ilgili bu bölümde temel işevleri sizlere aktardık. Sizde kendiniz örnekler yaparak bu bölümü tamamlayabilirsiniz. İleriki bölümlerde bu veri tabanları üzerine örnekler yapacağız.

14.8 Arama İşlevi (Search)

Veri tabanınız destekliyorsa arama yapabilmenizi kolaylaştıran *search* fonksiyonu kullanılabilir. Bir örnekle arama işlevini özetleyelim.

```
def baglanti = (database("internal","veritabani"))

def kullanici = (model(baglanti, "kullanicilar", {
    "id": "int",
    "isim": "text",
    "soyisim": "text",
    "kullanici": "text",
    "sifre": "text",
}))

// create fonksiyonu:
create(kullanici, {
    "id": "1",
    "isim": "Oytun",
    "soyisim": "Ozdemir",
    "kullanici": "oytun",
    "sifre": "1234"
})

//search fonksiyonu:
def sonuc = (search(kullanici, {"name": "oyt"}))
```

Burada *search* fonksiyonuna gönderilen parametrelere göre arama yapar. Sonuçlar bize karma dizi olarak dönmekoytunir.

Dip Not: **Her veri tabanı bu metodu desteklemeyebilir. Desteklemeksi için en indexleme özelliğinin veritabanında bulunması gerekir.**

Bölüm IV

Yardımcı İşlevler

Bölüm 15

Yardımcı İşlev Nedir?

Yardımcı işlevler genellikle bir programlama dilinde olamsı gereken yapılan işlevlere yardımcı olan özellikler içerir. Bu bölümde bu işlevlerin detaylarından bahsedeceğiz.

15.1 Dosya işleme (File Render)

O Dili üzerinde dosya içeriklerine bazı parametreler göndererek yeni bir değişken oluşturmanıza yardımcı olan dosya render işlemleri mevcuttur.

```
def sonuc = (renderf("dosya.txt", {...veriler...}))
```

Sonuç değişkenimize dosya.txt içeriğine veriler göndererek yeni bir metin oluşturduk. Değişkenler şu şekilde tanımlanabilir;

```
{{.parametre}}
```

Bu şekilde dışardan gelen veri *parametre* ile değiştirilir. Oluşan yeni metin istenildiği gibi kullanılabilir.

15.2 XML ile çalışmak (XML Parse)

XML dosyaları sadece url'den çektiğimiz XML içeriklerinde değil normal XML ile çalışırkende kullanabileceğimiz *xmlf* fonksiyonlarını içerir. XML (X Markup Language) dosyaları ile çalışırken *read* fonksiyonu kullanıp içeriğini alıp üzerinde fazla kod yazmamak için geliştirildi. XML parse etmek için *xmlf* fonksiyonu işinize yarayacaktır.

```
def sonuc = (xmlf("dosya.xml"))
```

Sonuç değişkenine *xmlf* fonksiyonu kullanarak dosya.xml dosyamı çektim ve tanımladık.

Eğer dosya değilde metini dönüştürmek isterseniz *xmlp* fonksiyonunu kullanabilirsiniz.

```
def sonuc = (xmlp("...xml..."))
```

Sonuç değişkenine *xmlp* fonksiyonu kullanarak xml ile yazılmış içeriği tanımladık.

15.3 JSON ile çalışma (JSON encode/decode)

JSON dosyaları ile çalışmak oldukça kolaydır. Bunun için basit bir kaç fonksiyonumuz mevcut. Bunlar; *jsonp* ve *jsonf* fonksiyonlarıdır.

```
//jsonp fonksiyonu:
def sonuc = (jsonp("json metni"))
//jsonf fonksiyonu:
def sonuc = (jsonf("dosya.json"))
```

jsonp fonksiyonu metin olarak gelen JSON içeriğini karma diziye dönüştürür ve sonuç olarak bir obje verir.

jsonf fonksiyonu bir JSOn dosyasını alarak otomatik olarak bir karma diziye dönüştürür ve sonuç olarak bir obje verir.

Bu iki fonksiyonu kullanarak sizde dosyadan yada metin olarak gelen JSON içeriklerini O Dili üzerinde kullanabilirsiniz.

15.4 Terminale renkli çıktı alma (Colorize)

Terminalden renkli çıktı almak için geliştirilmiş yardımcı bir fonksiyondur. Terminaliniz renkleri destekliyorsa renkli çıktı alabilirsiniz.

```
colorize("red", "Merhaba Renkler!")
colorize("blue", "Merhaba Renkler!")
colorize("green", "Merhaba Renkler!")
colorize("yellow", "Merhaba Renkler!")
colorize("grey", "Merhaba Renkler!")
colorize("black", "Merhaba Renkler!")
colorize("bold", "Merhaba Renkler!")
colorize("magenta", "Merhaba Renkler!")
colorize("white", "Merhaba Renkler!")
colorize("red", "Merhaba Renkler!", true)
colorize("blue", "Merhaba Renkler!", true)
colorize("green", "Merhaba Renkler!", true)
colorize("yellow", "Merhaba Renkler!", true)
colorize("grey", "Merhaba Renkler!", true)
colorize("black", "Merhaba Renkler!", true)
colorize("bold", "Merhaba Renkler!", true)
colorize("magenta", "Merhaba Renkler!", true)
colorize("white", "Merhaba Renkler!", true);
```

Renkleri doğrudan terminalinizde kullanabilirsiniz.

15.5 Düzenli İfadeler (Regular Expressions)

Düzenli ifadeler günümüzde karmaşık metinlerde işlem yapmayı kolaylaştıran parametre grupları olarak nitelendirilebilir. Bu parametreleri O dili üzerinde nasıl işlendiğini görelim.

```
def rex = "[A-Z][a-z]"
def sonuc = regexp_check(rex, "Merhaba!")
true
```

check metodu bir metinde verilen düzenli ifadenin var olup olmadığını doğrular. Sonuç olarak bize doğru yada hatalı olarak dönüş sağlar.

```
def rex = "ba"
def sonuc = regexp_find(rex, "Merhaba!")
["ba"]
```

find fonksiyonu ise içinde geçenleri bize bulur ve dizi olarak döndürür. Bu şekilde düzenli ifadeleri kullanarak metin üzerinde ayıklama ve doğrulama yapabiliriz.

15.6 Şifreleme (Encrypt/Decrypt)

Bir metni şifrelemek yada bir içeriği şifrelemek için kullanılan bir yardımcı fonksiyonlardır. Bu şifreleme işlevleri metni şifreler ve gizlemeye yardımcı olur.

```
// şifreleme örnek:
def sifrele = (encrypt("anahtar", "metin"))

// çözme örnek:
def coz = (decrypt("anahtar", sonuc))
```

Örnekteki gibi *sifrele* degiskenimize anahtar kullanarak metni şifreledik. *sifrele* değişkeni burada kullanılamaz veya çıktısı alınamaz. Bir dosyaya yazarak saklayabiliriz. İçeriği görüntülenmediği için çözmek gerekebilir. Çözmek için ikinci *decrypt* fonksiyonunda anahtarı kullanarak metni çözebiliriz ve metine ulaşabiliriz. Kullanımı gördüğünüz gibi oldukça kolaydır.

15.7 Metin Değiştirme (Replace)

Metinler üzerinde çalışma yaparken metni bulup değiştirmek için replace işlevi kullanılır. Bu işlev metin üzerinde belirtilen metni bulur ve değiştirir.

```
def metin = "Merhaba ben corç!"
def metin = (replace(metin, "corç", "borç"))
println(metin)

"Merhaba ben borç!"
```

Gördüğünüz üzere *replace* kullanarak "corç" kelimesini "borç" ile değiştirdik. Sizde kendi testlerinizi yapabilirsiniz.

Bölüm V

Tehlikeli Sular (Biohazard)

Bölüm 16

Neden Tehlikeli?

Bu kısmın kritik olmasının ve en sonunda anlatmamın amacı bu kritik işlemlerin sistem tarafında doğrudan işliyor olması ve bu kısımda yapaccağınız işlemler sisteminize zarar verebilir. Eğer çok düşük seviye programlama işlevlerine ihtiyacınız varsa buradaki işlemler sistem tarafında fiziksel çalışmalar yapmanızı sağlayabilir.

Dip Not: **Bu çalışmalar çok düşük seviye fonksiyonlar ve alt seviye programlama bilgisiside gerekmekoytunir.**

Bölüm 17

Metin Olarak Kod Çalıştırmak (Eval)

Bu çok kritik durumlarda kullanılabilir. Metin olarak bir kodu *eval* fonksiyonu kullanılarak bir metni doğrudan çalıştırılmasını sağlayabilirsiniz. Bu fonksiyonun zararlı yanı eğer şifrelenmiş bir içerik varsa onu *decrypt* ederek çalıştırılması sağlanabilir. **Bu durumda bilinmeyen kodların çağrılması sisteme zarar verebilmekoytunir.**

Tehlikeli bir örnek vereceğiz. Bu örnek sadece olayı daha detaylı anlamanız açısından faydalı olacaktır.

```
def code = "def x = 1";
(eval(code))
println(x)
```

```
1
```

Örnekte gördüğünüz gibi *code* değişkenimiz bir kod içermekoytunir. *Eval* ile bunu çalıştırdık. İçindeki kod otomatik çalıştı ve x değişkeni tanımlandı.

Şimdi sizlere daha tehlikeli bir örnek vereceğiz.

```
// key.txt:
111-222-444
```

```
// source.ola:
def x = 1
```

```
// sifreleme.ola:
def key = (read("key.txt"))
def source = (read("source.ola"))
def dosya = (encrypt(key, source))
write("sifreli.ola", dosya)
```

```
// coz.ola:
def key = (read("key.txt"))
def dosya = (read("sifreli.ola"))
def coz = (decrypt(key, dosya))
eval(coz)
println(x)
```

```
1
```

Şimdi bu örneği açıklayalım. Anahtarı tutması için *key.txt* dosyası yaptık. Bu dosya dosyayı açacak anahtarı taşıyacak. *source.ola* dosyamız şifrelenecek kodları içerecek. *sifreleme.ola* şifreleme programımız olacak. *coz.ola* programımız şifreyi çözecek ve kodu çalıştıracak. Şifreyi çözerken *key.txt* dosyasını kullanacak. Hiç bir şekilde *sifreli.ola* dosyası okunamayacağından kullanıcı kodun kaynağını görmeyecek. Bu şekilde dosya şifrelenmiş olarak çalışacak.

Bu kısımda tehlike tam olarak *sifreli.ola* dosyasında bulunmaktadır. Bu kod şifrelendiği için tehlikeli bir kaynak kodu içerebilir. Bu durum bilgisayara zarar verebilir yada hasar oluşturabilir. Yada çözülüp başkası tarafından kod eklenip tekrar şifrelenebilir. Bu tehlikeli bir süreçtir.

Bölüm 18

Makine Operasyonları (Assembly)

Doğrudan Assembly operatörlerini çalıştırmak aşırı riskli bir durumdur. Makinada doğrudan sanal bir işlemci üzerinde bu kodlar işletilir ve makina doğrudan fiziksel olarak bu kodları işletir. **Bu kısımddaki kodları çalıştırmak son derece tehlikeli sonuçlar sağlayabilir. Eğer Assembly bilginiz yoksa önce bu kodların neyi çalıştırdığını bilmelisiniz.**

Burada sadece nasıl çalıştırıldığını göstereceğiz. Bu nedenle örnek vermek yerine bu kodları siz kendi tarafınızda test etmelisiniz. Bu operatörleri kullanmak için *asm* fonksiyonu kullanılır.

```
asm(operatorkodu, degisken islevi, parametreler...)
```

Örnek bir hello world yaparak kodumuzu test edelim.

```
def hello = "Hello!";
def len = len(hello)

asm("DB", "0xa", hello)
asm("EQU", len)
asm("MOV", "edx", len)
asm("MOV", "ecx", msg)
asm("MOV", "ebx", 1)
asm("MOV", "eax", 4)
asm("INT", "0x80")
asm("MOV", "eax", 1)
asm("INT", "0x80")
```

```
Hello!
```

Assembly DB kullanarak bellekteki *0xa* adresine "Hello!" değişkenimizi yazdık. Boyutunu hesaplayıp bellekteki *0xa* adresinden başlayıp bu değere gelene kadar bu mesajın yer alacağını belirttik. Tüm bu değerleri *edx* ve *ecx* adreslerine taşıdık. *ebx* bloğuna 1 gönderdik. Bunun anlamı sistem çıktısı ver demektir. *eax* adresine 4 gönderdik. Bunun anlamı sisteme bu data bloğunu yani bellekteki *0xa* değerinde ve *equ* boyutunda data yaz demektir. *int* bloğuna *0x80*

gönderdik. Sistemden çağırma işlevi taşımaktadır. *eax* adresine 1 gönderdik. ve tekrar çağırma işlevini çalıştırdık. Sisteme gönderilen değişkeni çağır ve sisteme çıkış sinyali gönder anlamındadır.

Bu şekilde makina operasyonları çalıştırdık. Düşük seviyeli işlemciye makina kodu işleyerek sonuç döndürmesini isoytunik.

Bölüm 19

Doğrudan komut çalıştırmak (Open/Run/Exec)

Komut çalıştırmak aslında tehlikeli olmayabilir. Fakat bazen zararlı yazılımlar çalıştırmak için buradaki fonksiyonlar kullanılabilir. **Bu komutlar sistemimize zarar verebilir.** Biz yinede öğrenmenizi istiyoruz. Bu nedenle bunların işlevlerini bu bölümde özetleyeceğiz.

19.1 Doğrudan Açma Fonksiyonu (Open)

Açma komutu doğrudan sistemde program açılmasına ve çalıştırılmasını sağlar. Bu komutu kullanarak sistemde uygulama açabilirsiniz.

```
open("program", ...parametreler...)
```

Program ve parametreleri kullanarak uygulama açabilirsiniz.

```
open("firefox")
```

Örnekteki gibi firefox'u çalıştırdık. Eğer sisteminiz arayüz çalıştırmaya uygunsa firefox çalışacaktır.

19.2 Komut Çalıştırma (Run)

Komut çalıştırma işlevi farklı komutlar çalıştırabilmenizi sağlar. **İşlem oluşturmadan komut çalıştırmanız durumunda çalışan komutları kontrol edemeyebilirsiniz.**

```
run("program", ...parametreler...)
```

Örnek olarak bir program çalıştıralım ve sonuçlarını bir değişkene eşitleyelim.

```
def sonuc = (run("ls", "-la"))
```

Listeleme komutunu çalıştırdık ve *sonuc* değişkenine eşitledik.

19.3 Sistem Komutu Çalıştırma (Exec [SH/CMD])

Hangi sistemde çalıştığınızı biliyorsanız, sisteme uygun çalıştırma işlevlerini başlatıp; sonuçlarını bir değişkene eşitleyebilirsiniz.

sh sadece linux ve unix tipi sistemlerde çalışır:

```
def sonuc = (sh_exec("ls", "-la"))
```

cmd sadece windows tipi sistemlerde çalışır:

```
def sonuc = (cmd_exec("dir", "/w"))
```

Bu şekilde çalıştığınız platforma özel komutlar işletebilirsiniz. İki *sonuc* değişkenide listeleme komutlarını çalıştırır. Bu şekilde sizde kendi komutlarınızı işletebilirsiniz.

Dip Not: **Bu şekilde işlem açmadan program yada komut çalıştırmak son derece kontrolsüz şekilde işlem yapmak gibidir. Sisteminize zarar verebilirsiniz. İşlem açarak komut çalıştırmanız 7.3 de belirttiğimiz şekilde daha uygun olacaktır.**

Bölüm VI

Örnek Uygulamalar

Bölüm 20

Web çatısı oluşturmak (Web Framework)

Web Çatıları, bir web sitesini oluştururken yada çalıştırırken, çeşitli yardımcı fonksiyonlar içeren yapılardır. Bütün olarak yazılan kodları insanların görüntüleyeceği şekilde bir web sitesine dönüştürürler.

Web çatısı yapmak için öncelikle nasıl bir klasörleme yapısı yapacağınızı belirleyin. Basit olarak başlayalım klasör yapısı ve diğer yapıyı belirleyelim.

```
boot.ola
config
controllers
database
library.olp
proc.ops
public
templates
vendor
```

Programı boot.ola dosyası ile çalıştıracağız. Config dosyalarını okuyarak, Controllers dosyalarını dahil ederek; Templateler üzerinden render işlemlerini gerçekleştirecek. Config üzerindeki yönlendirmelere uygun bir biçimde sitede dolaşan insana sayfaları görüntüleyecek. Bağımsız olarak vendor altında dışarıdan indirilen kütüphaneler tutulur. Public klasöründe ise herkese açık dosyalar listelenecektir. Şimdi Uygulamamızın kaynak kodlarını inceleyelim. Bu kaynak kodları indirebileceğiniz adresi çalışmanın sonunda paylaşacağız.

```
// boot.ola:
# Controllers

load "vendor/ondle.ola"
load "controllers/index.olm"

# Configuration Definitions
load "config/config.ola"
```

```
println("Server running on 0.0.0.0:80.")
webserver("80", config)
```

Boot dosyamız vendor altında ondle.ola dosyası, controller/index.olm modülü, config/config.ola ayar dosyalarını dahil ederek 80 portu üzerinde bir web sunucusu başlatır.

```
// config/config.ola:
def config = {
  "/": {
    "type": "GET",
    "response": (main())
  }
};
def db = (database("internal","database"));
```

Config dosyamızda yönlendirme ve veri tabanı değişkenleri bulunmaktadır.

```
// controllers/index.olm:
def main = fn(){
  return (renderf("templates/index.html",
    {
      "title": "Olang Web Framework | Index",
      "body": "Welcome to the Olang Web Framework."
    }
  ));
};
```

Index controller modülü templates/index.html dosyasına title ve body değişkenlerini yazan, bir main tanımlaması yapar.

```
// library.olp
{
        "package": "olf",
        "publisher": {
                "name": "Oytun",
                "email": "info@oytun.org",
                "github": ""
        },
        "sources": "src",
        "main": "boot.ola",
        "requires": [
                {
                        "name": "olang-html",
                        "src": "https://gitlab.com/olanguage/libraries/olang-html.git"
                }
        ],
        "compiled": "",
        "install": null,
        "jobs": [
                {
```

```
                        "name": "start-web",
                        "command": "olang process --start true"
                },
                {
                        "name": "stop-web",
                        "command": "olang process --stop true"
                }
        ],
        "type": "project"
}
```

Kütüphanelerin ve gereksinimlerin yer aldığı library.olp dosyası,

```
$ olang install
[!] INFO: "vendor" path not found. Path creating...[ready]
[>] Fetching    [olang-html][done]
[>] Compiling   [vendor/olang-html/src/main.ola]
[>] Compiling   [/home/ted/workspace/olf1/vendor/olang-html/src/component.ola]
[>] Compiling   [/home/ted/workspace/olf1/vendor/olang-html/src/html.ola]
[>] Compiling   [/home/ted/workspace/olf1/vendor/olang-html/src/form.ola][done]
[>] Writing Autoload... [vendor/ondle.ola][done]
[X] Fetch Completed.
```

kurulum komutu çalıştırıldığında otomatik olarak vendor altına kütüphaneleri yükler.

```
// proc.ops
[olang]
boot: boot.ola
```

```
$ olang process --start true
2020/08/15 18:35:06 Running Proc: 5874
2020/08/15 18:35:06 open nohup.out: no such file or directory
```

Proc.ops dosyamız hangi işlemin hangi isimle işlem olarak arkaplanda çalıştırılacağını belirtir. Process komutunu çağırdığımızda otomatik olarak bu dosyaya bağlı işlevleri çalıştıracak ve web sitemiz kullanıma açılacaktır.