Kaan Kuşcu

ÖNSÖZ

Go programlama dilini anlatan kitabımızın 4. sürümüne hoşgeldiniz.

Kitabın yazımı süresi boyunca faydası dokunan insanlara çok teşekkür ediyorum. Tabi ki eleştiride bulunan insanlara da teşekkür ediyorum. Yaklaşık olarak 2 senedir zaman buldukça ücretsiz olan bu kaynağı geliştirmeye çalışıyorum. Bu kitap vesilesi ile Go ile ilgili bir sürü arkadaşım oldu. Kendileri ile fikir alış-verişinde bulunduk. Bu fikirler doğrultusunda kitabı ilerletmeye devam ediyor olacağım. Önsöz kısmını kısa tutmak istiyorum. Çünkü burayı okumak çoğu kişinin hoşuna gitmiyor :)

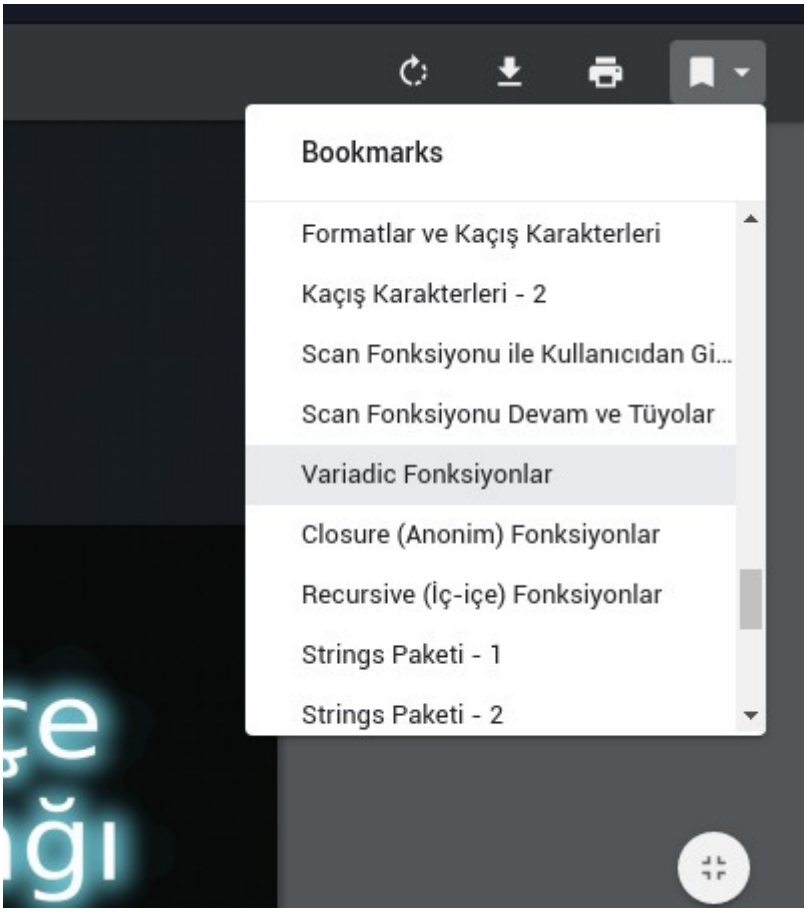
Kitap hakkında veya Go programlama dili hakkında danışmak istediğiniz bir konu varsa kitabın son sayfasında bulunan bölümden iletişim bilgilerime ulaşabilirsiniz.

Eğer sizde bu kaynağın geliştirilmesinde rol oynamak istiyorsanız, benimle (Kaan Kuşcu) iletişim kurabilirsiniz.

Yardımcı olacak veya eleştiride bulunacak kişilere şimdiden teşekkür ederim.

Daha iyi bir okuma deneyimi için Google Chrome ile okumanızı tavsiye ederim.

İçindekiler listesi için sağ üstteki Bookmark butonunu kullanabilirsiniz.



Giriş

Bu eğitim kaynağında Golang programlama dili hakkında bilgilerdirici ön yazı, kullanım şekline ve örneklerine bakacağız. Bu kitapla pratik yapabilirsiniz. Kitap ileri seviye Go programlama içermeyecektir.

Bu Kitap Kimlere Hitap Ediyor?

Bu kitap;
- Go programlama dilini öğrenmek isteyen,
- Go'da giriş seviyesinde bilgi sahibi olan,
- Go'da orta seviyede bilgi sahibi olan,
- veya daha önce başka dillere aşina olan,

kişilere hitap ediyor

Amaç

Kitabın Amacı;
- Golang için Türkçe kaynak oluşturmak
- Golang için ücretsiz eğitim kaynağı oluşturmak
- Golang dilinin temel yapısını öğretmek

Kitabın İçeriği Hakkında

Kitabın içeriğinde Go programlama dilinden "Go", "Golang" ve "Go Programlama Dili" olarak bahsediyor olacağım. Hepsi aynı anlama geliyor. Genellikle kodların kullanım şekli ve yapısından bahsediyor olacağım. Tabi ki işleyişi anlayabilmemiz için örnekler ile pekiştireceğim.

Bu kitap sayesinde ülkemizde şuanda diğer dillere göre daha az bilinen ve hakkında fikir sahibi olan kişilerin sayısı az olan bu programlama dili hakkında kaynak oluşturmak istiyorum.

Ücretli bir kitap olduğu zaman ilgi görmeyeceğinden dolayı ücretsiz olarak PDF şeklinde yayınlamaya karar verdim. Böylece Go diline ilgiyi çekmeyi hedefliyorum.

Go Programlama Dili Hakkında Kişisel Görüşüm

Go programlama dilini aslında 2018 yılında farkında oldum. Daha önceden VB .Net, Python, Java ve C ile uğraşıyordum. Kendi görüşüme göre bana derlenebilir diller daha yakın geliyordu. Ama C'de bazen insanı çileden çıkarıyordu.

Daha kendimi C++'ta ilerletmeye başladım. C++ güzel bir dil olmasına rağmen bir türlü ısınamadım. Haliyle de yeni bir dil arayışına çıktım. Aslında Back-end'den daha fazla Front-end geçmişim vardır. Bu yüzden de iki tarafada hizmet edilebilen bir dil araştırıyordum ve aynı zamanda derlenebilen.

İnternette birkaç video izledikten sonra Go'yu gördüm. Sözdizimi olarak gayet basit kuralları olan bir dildi. Sanki derlenebilir bir Python gibiydi. Paket yönetim sistemi gözüme iyi geldi. Fakat Türkçe kaynak sıkıntısı olan bir dildi.

Daha sonra İngilizce (ve bazen İspanyolca) kaynaklardan Go'yu öğrenmeye başladım. Öğrendikçe ne kadar programcı odaklı bir dil olduğunu anladım. Zaten Go'yu geliştiren adamlara bakınca bu işin içinden geldiğini anlıyorsunuz.

Özet olarak Go benim için yazımı kolay, anlaması kolay, paket yönetimi kolay ve hızlı derlenen bir dildir.

Kitap Yazma Fikri

Başta Golang'i öğrenmek için bilgisayarıma ufak tefek notlar alıyordum. Daha sonra notlarımı düzenli olarak biryere kaydediyordum. Son olarak baktığımda elimde Go'nun temelini neredeyse anlatacak bir kaynak biriktiğini farkettim. Daha sonra eksikleri tamamlayarak Go öğrenmek isteyen kişilerin ihtiyacını giderebilmesi için bir kaynak oluşturma kararı aldım.

İmla kurallarına uygun ve doğru bilgiler içerin bir kaynak oluşturmak için çabaladım. Kaynağı ilk olarak vaktimin çoğu geçirdiğim bir forumda tanıttım. Gerçekten güzel tepkiler ile karşılaştım. Arada tabi kötü tepkilerde aldım. Bazen motivasyonumu düşürecek tepkiler aldım.

Sosyal medya hesaplarımdan teşekkür mesajları da aldım. Kendilerine bana verdikleri motivasyon için teşekkür ediyorum.

Sonuçta bu kitaptan bir gelir kazanmıyorum. İnsanların yazdığım bir şey hakkında yorum yapması hoşuma gidiyor :)

Siz de yorumlarınızı esirgemezseniz sevinirim...

Golang Hakkında

Golang (diğer adıyla Go), **Google**'ın **2007** yılından beri geliştirdiği açık kaynaklı programlama dilidir. Daha çok alt-sistem programlama için tasarlanmış olup, derlenebilir ve **statik** tipli bir dildir. İlk versiyonu **Kasım 2009**'da çıkmıştır. Derleyicisi olan **"gc" (Go Compiler)** açık kaynak olarak birçok işletim sistemi için geliştirilmiştir.

Robert Griesemer

Rob Pike

Ken Thompson

Golang, Google mühendislerinden olan **Robert Griesemer, Rob Pike ve Ken Thompson** tarafından ilk olarak deney amaçlı ortaya çıkmıştır. Diğer dillerdeki eleştirilen sorunlara çözüm getirmek ve iyi yönlerini korumak için çıkarılmıştır.

Bu adamların daha önceden bulunmuş olduğu projelere bakacak olursak Google'ın gerçekten bu kişileri cımbızla seçtiğini anlayabiliriz. İşte Golang'ın yaratıcılarının bulunduğu projeler:

Robert Griesemer: Hotspot ve JVM (Java Sanal Makinesi)
Rob Pike: UNIX ve UTF-8
Ken Thompson: B, C, UNIX ve UTF-8

Dilin Özellikleri

- Statik yazılmıştır, fakat dinamik rahatlığındadır.
- Büyük sistemlere ölçeklenebilir.
- Üretken ve okunabilir olması, çok fazla zorunlu anahtar kelime ve tekrarlamaların kullanılmaması
- Tümleşik Geliştirme Ortamına (IDE) ihtiyaç duyulmaması fakat desteklemesi
- Ağ ve çoklu işlemleri desteklemesi
- Değişken tanımında tür belirtimi isteğe bağlıdır.
- Hızlı derlenme süresi
- Uzak paket yöneticisi (go get)

Örnek Merhaba Dünya Uygulaması

```
package main

import "fmt"

func main(){
        fmt.Println("Merhaba Dünya!")
}
```

Bilgisayarınız Üzerinde Golang Geliştirme

Öncelikle bilgisayarımız üzerinde nasıl Golang geliştireceğimize bakalım. Geliştirmek derken Golang programı oluşturacağımızı kastediyorum. Öncelikle Golang'ın resmi sitesinden Golang programını indiriyoruz.

Buradan İndirebilirsiniz
https://golang.org/dl/

Golang'ın basit bir kurulumu var o yüzden kurulumu atlıyorum.

Linux İS kullananlara tavsiyem, kullandığınız dağıtımın uygulama deposundan Golang'ı indirin. Sizin için daha kolay olur.
Golang'ı indirdiğimize göre bize Golang kodlarımızı yazacağımız bir Tümleşik Geliştirme Ortamı (IDE) lazım. IDE'ler kodlarımızı yazarken kodların doğruluğunu kontrol eder ve kod yazarken önerilerde bulunur. Bu da kod yazarken işimizi kolaylaştırır.

Benim tavsiyem çoğu kodlama dilini yazarken kullandığım ve Golang yazanların da popüler olarak kullandığı **Visual Studio Code** programı.

Buradan İndirebilirsiniz
https://code.visualstudio.com/Download

Linux İS kullananlara yine kullandıkları dağıtımın uygulama deposundan indirmelerini tavsiye ediyorum.

Visual Studio Code'dan ilerki zamanlarda **vscode** olarak bahsedeceğim.

Go eklentisinin düzgün bir şekilde kurulabilmesi için bilgisayarımızda **git** komut-satırı uygulaması bulunması gerekir. Çünkü eklentinin yüklenmesinden sonra Go eklentisi VSCode için 15 civarı aracı otomatik indirecek. Git'in yüklü olup olmadığını öğrenmek için komut satırına aşağıdakileri yazın:

```
git --version
```

Eğer versiyon numarasını gördüyseniz yüklü demektir. Eğer yüklü değilse veya git'i güncellemek istiyorsanız ki, mutlaka öneririm, aşağıda nasıl yükleneceğini görebilirsiniz.

Windows
Buradan İndirebilirsiniz.

MacOS
Buradan İndirebilirsiniz.

GNU/Linux İşletim Sistemleri
Debian/Ubuntu
sudo apt-get install git

Fedora
dnf install git

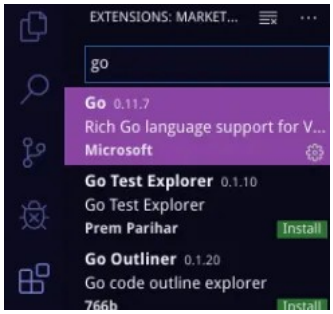
Arch Linux
sudo pacman -S git

Gentoo
emerge --ask --verbose dev-vcs/git

openSUSE
zypper install git

Mageia
urpmi git

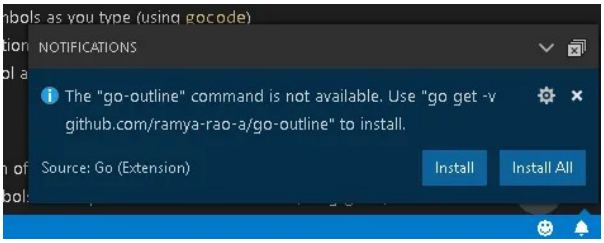
Git kurulumunu da yaptığımıza göre VSCode için Go eklentisini kurabiliriz.



Vscode'un sol tarafından **Extension** (Eklentiler) sekmesine geçiyoruz. Arama kutusuna **go** yazıyoruz. Resimde de seçili olan Go eklentisini yeşil **Install (Yükle)** butonuna basarak yüklüyoruz. Yeniden başlatma isterse başlatmayı unutmayın.

Daha sonra **main.go** adında bir dosya oluşturup VSCode ile açalım. main.go dosyasının içerisine rastgele birşeyler yazdığımızda VSCode sağ alt tarafta bize uyarı verecektir.

Install All diyerek Go eklentisi araçlarının kurulumunu başlatalım.



Kurulum bize 15 civarı araç kuracak. Başarı ile kurulan araçların yanında **SUCCEED** ibaresi yer alır.

Tüm araçlarımız başarıyla kurulunca artık VSCode üzerinden Go geliştirmeye hazır olacaksınız.

Merhaba Dünya

Programlama dünyasında gelenektir, bir programlama dili öğrenilirken ilk önce ekrana **“Merhaba Dünya”** çıktısı veren bir program yazılır. Biz de geleneği bozmadan Golang üzerinde Merhaba Dünya uygulaması yazalım. İlk önce kodları görelim. Daha sonra açıklamasını görelim.

```
package main

import “fmt”

func main(){
	fmt.Println(“Merhaba Dünya!”)
}
```

Şimdi yukarıdaki kodlarda neler yaptığımıza gelelim.

Package, kod sayfalarımız arasında iletişimde bulunabilmemizi sağlar. Bu sayede içerisinde **package** değeri aynı olan kod sayfaları birbirleriyle iletişim halinde olma yeteneği kazanır. Yukarıdaki örnekte package uygulama olan sayfalar birbiriyle iletişim kurabilir.

import “fmt” ile Golang dilinde kullanılan ana işlemler için olan kütüphanemizi içeri aktardık.

func main() ise programımızın çalışacağı ana bölümün fonksiyonudur. Yanındaki süslü parantezler { } içine yazdığımız kodlar ile programımızda çeşitli işlemler yapabileceğiz. Derlenmiş bir uygulama ilk olarak **main** fonksiyonuna bakar ve buradaki kodları çalıştırır.

Yukarıda **import** ettiğimiz **fmt** kütüphanesi içinden **Println** fonksiyonu ile ekranımıza **“Merhaba Dünya”** yazısını bastırdık. Gelelim programımızın derlenmesine. Daha önceden programlama dilleriyle geçmişi olmayan arkadaşlarımız için derlenme şöyle anlatılabilir. Yazdığımız Golang dili insanların kolaylıkla programlama yapabilmesi için oluşturulmuş bir dildir.

Ama makine (bilgisayar) bu yazdıklarımızı anlamaz. O yüzden her derlenen dilde olduğu gibi Golang’ın da yazdıklarımızı makinenin anlayacağı makine diline çeviren derleyicisi vardır.

Makinemiz üzerinde çalıştırılabilir bir dosya üretmek için kodlarımızın derlenmesi gereklidir. Vscode üzerinden kodlarımızın derlenip çalışması için **F5** tuşuna basıyoruz. Böylece programımızı test edebiliriz. Eğer vscode üzerinden derliyorsanız yazdığımız kodların hemen altında **DEBUG CONSOLE** bölümünde kodumuzun sonuç çıktısını görebiliriz.

Çıktımızı inceleyecek olursak, **API server listening at :127.0.0.1:44830** ibaresinde gerçekleşen olay, Golang kodlarımızı çalıştırdığımızda oluşturulan **44830** portlu **127.0.0.1** yerel sunucusu (localhost) üzerinden kodlarımızın sürüş testini gerçekleştirdik. Hemen aşağısına da çıktımızı verdi.

Eğer vscode üzerinden değil de, konsol üzerinden yapmak isterseniz, oluşturmuş olduğumuz main.go dosyasının bulunduğu dizin (klasör) içerisinde konsol uygulamamızı açıyoruz. Windows’ta cmd veya Powershell’dir. Unix sistemlerde terminal diye geçer. İki tarafa da yazacağımız komutlar aynıdır. O yüzden hangisinden yazdığınız farketmez.

Kodlarımızı sadece denemek istiyorsak yazacağımız komut::

```
go run main.go //main.go yerine .go dosyamızın ismi gelecek
```

Eğer çalıştırılabilir bir dosya oluşturmak istiyorsak (Windows’ta .exe)

```
go build main.go //main.go yerine .go dosyamızın ismi gelecek
```

Böylece bu işlemleri yaptığımız dizin içerisine çalıştırılabilir bir dosya oluşturmuş olduk.

Windows üzerinden konsola **main** yazarak uygulamayı açabilir veya klasörde **main.exe** dosyasına çift tıklayarak uygulamayı çalıştırabilirsiniz.

Linux üzerinden ise terminale derlenmiş main çıktınızın bulunduğu dizinde **./main** yazarak çalıştırabilirsiniz.

Böylece ilk uygulamamızı yazmış olduk. Tabi şu ana kadar görmemiş olduğumuz kodlar gördük. Onların açıklamaları da ileriki bölümlerde olacak. Şimdilik gelenek diye ilk bölümde **Merhaba Dünya** uygulaması yazdık.

Farklı Platformlara Build (İnşa) Etme

Golang projemizi build (inşa) ederken, yani çalıştırılabilir bir dosya üretirken **go build dosya.go** şeklinde build ederiz. Bu işlem varsayılan olarak kullanmakta olduğumuz işletim sistemi için build işlemi yapar. Yani Windows kullanıyorsak Windows'ta çalışmak üzere Linux İS kullanıyorsak Linux İS'te çalışmak üzere dosya oluşturur. Aynı şekilde sistemimizin mimarisi 32bit ise 32bit için, 64bit ise 64bit için çalıştırılabilir dosya üretir. Örnek olarak sistemimiz Windows 64bit ise oluşturduğumuz çalıştırılabilir dosya (exe dosyası) sadece Windows 64bitlerde çalışır.

Eğer farklı işletim sistemi için bir çalıştırılabilir dosya üretmek istiyorsak aşağıdaki komutu kullanmamız gerekir.

```
env GOOS=hedef-sistem GOARCH=hedef-mimari go build hedef-dosya
```

Örnek olarak, **main.go** dosyamızı **Windows 32bit** sistemler için build etmek istersek aşağıdaki komutları girmemiz gerekir.

```
env GOOS=windows GOARCH=386 go build main.go
```

Bu işlem ile main.exe adında bir dosya elde ederiz. Bu dosya Windows 32bit sistemlerde çalışabilir. Biliyorsunuz ki 32bit uygulamalar 64bit sistemlerde çalışır ;fakat 64bit uygulamalar 32bit sistemlerde çalışmaz. Onun için bir uygulama build ediyorken bunu aklınızdan çıkarmayın. Golang'ın dosya build edebildiği işletim sistemi ve mimari sayısı oldukça fazladır.

Golang'ın build edebildiği tüm işletim sistemi ve mimarilerine bakmak gerekir ise;

GOOS	GOARCH
android	arm
darwin	386
darwin	amd64
darwin	arm
darwin	arm64
dragonfly	amd64
freebsd	386
freebsd	amd64
freebsd	arm
linux	386
linux	amd64
linux	arm
linux	arm64
linux	ppc64
linux	ppc64le
linux	mips
linux	mipsle
linux	mips64
linux	mips64le
netbsd	386
netbsd	amd64
netbsd	arm
openbsd	386
openbsd	amd64
openbsd	arm
plan9	386
plan9	amd64
solaris	amd64
windows	386
windows	amd64

* Tablo harf sıralamasına göre yapılmıştır.

Birkaç örnek daha yapmak gerekirse;

Windows 64bit Build

```
env GOOS=windows GOARCH=amd64 go build main.go
```

Linux 32bit Build

```
env GOOS=linux GOARCH=386 go build main.go
```

MacOS 64bit Build

```
env GOOS=darwin GOARCH=amd64 go build main.go
```

Klasör Build (İnşa) Etme

Oluşturduğumuz **.go** dosyaları birden fazla parçadan oluşuyorsa aşağıdaki komut ile klasör içerisindeyken build işlemini yapabiliriz.

```
go build . //sonda nokta işareti var
```

Eğer klasörün dışından build işlemi yapacaksak nokta yerine klasörün yolunu girmemiz gerekir. Aşağıda gösterilmiştir.

```
go build /klasör/yolunu/buraya/girin
```

Build işleminde sadece **.go** dosyaları derlenir. Tüm dosyalar işlenip tek başına çalıştırılabilir dosyaya dönüştürülür. Yani yanındaki Html, Css, Js vs. türünde dosyalar paket içine alınmaz. Tümüyle dosyları paketlemek için ek kütüphaneler kullanabilirsiniz. Önerim **Statik** isimli kütüphanedir. İlerideki bölümlerde zaten bu kütüphaneyi kullanıyor olacağız.

Paketler

Her Golang programı paketlerden oluşur ve kendisi de bir pakettir.

```
package uygulama
import "fmt"
func main() {
   fmt.Println("Merhaba Dünya") // Çıktımız
}
```

package terimi ile programımızın paket adını belirleriz. Hemen aşağısında da **import "fmt"** ile **fmt** paketini çektiğimizi görebilirsiniz. Yani çektiğimiz fmt paketi de bir programdır. Bilin bakalım fmt paketinin ilk satırında ne yazıyor? Tabiki de **package fmt**. Yani package ile programımızın ismini tanımlıyoruz. Bu ismi kullanarak diğer paketler ile iletişimde bulunabiliriz.
import terimi ise yazıldığı pakete başka bir paketten bir yetenek aktarmaya yarar. Yetenekten kastım, import edilen paketin içinde fonksiyonlar mı var? Struct'lar mı var? vs. onları içeri aktarır.

```
import (
   "fmt"
   "math"
)
```

Yukarıda birden fazla paket import etmeyi görüyoruz. math paketi bize ileri matematiksel işlemler yapabilmemiz için gerekli fonksiyonları sağlar.

Yorum Satırı

Diğer dillerde de olduğu gibi Golang'ta yorum satırı özelliği mevcuttur. Yorum satırı derleyici tarafından işlenmez. Yani görmezden gelinir. Bu bölüme kendiniz için açıklama vs. bilgiler yazabilirsiniz. Golang'ta yorum satırı oluşturmak için 2 yöntem mevcuttur.

// Çift Taksim Yöntemi
Bu yöntem ile derlenmesini istemediğimiz yazının başına çift taksim ekleyerek görmezden gelinmesini sağlıyoruz.

```
//Buraya herhangi birşey yazabilirsiniz
```

/* */ Taksim-Yıldız Yöntemi
Bu yöntem ile birden fazla satırın derlemede görmezden gelinmesini sağlayabiliriz.

```
/* Buraya
herhangi
birşey
yazabilirsiniz */
```

Veri Tipleri

Integer Türler
Öncelikle tüm integer türleri bir görelim;
int, int8, int16, int32, int64
uint, uint8, uint16, uint32, uint64, uintptr

Bu veri tipleri içerisinde sayısal değerleri depolayabiliriz. Fakat şunu da unutmamalıyız. Her sayısal veri tipinin depolayabildiği maksimum bit vardır. Örnek olarak uint8 veri tipinin maksimum uzunluğu 8 bit'tir. Bitler 0 ve 1 sayılarından oluşur. 8 bit demek 8 haneli 1 ve 0 sayısı demektir. Int8 maksimum alabileceği sayı derken 11111111 (8 tane 1), yani onluk sistemde 255 sayısına denk gelir. int 8 ise pozitif olarak +127, negatif olarak -128 maksimum değerinin alabilir. (127+128=255). int16 +32767 ve -32768 maksimum değerlerini alır. Int32 +2147483647 ve -2147483648 maksimum değerlerini alır. Int64 +9223372036854775807 ve -9223372036854775808 maksimum değerini alır.

U harfi ile başlayan sayı veritiplerinde ise sayının değeri pozitif veya negatif işarette değildir. Sadece bir sayısal değerdir. U'nun anlamı unassigned yani işaretsizdir. Uint8 0-255 arası, uint16 0-65535, uint32 0-42967295 arası, uint64 0-18446744073709551615 arası değerler alabilir. Uintptr ise yazdığınız sayıya göre alanı belirlenir.

Integer sayısal veri tipleri içierisinden bahsedebileceğimiz son tipler ise int ve uint. Int ve uint veri tipleri kullanmış olduğumuz işletim sistemi 32bit ise 32bit değer alırlar, 64bit ise 64bit değer alırlar. Sayısal bir değer atanacağı zaman en çok kullanılan veri tipleridir. Genellikle int daha çok kullanılır. Eğer çok meşakkatli bir program yazmayacaksanız int kullanmanız önerilir.

Byte Veri Tipi: uint8 ile aynıdır.
Rune: int32 ile aynıdır. Unicode karakter kodlarını ifade eder.

Float Türler
Float türleri integer türlerden farklı olarak küsüratlı sayıları tutar. Örnek: 3.14
Lütfen Dikkat!

Küsüratlı sayılar İngiliz-Amerikan sayı sistemine göre nokta koyarak ifade edilir. Türk sistemindeki gibi virgül (3,14) ile ifade edilmez.

float32: 32bitlik değer alabilir.
float64: 64 değer alabilir.

Complex Türler
Complex türleri içerisinde gerçel küsüratlı (float) ve sanal sayılar barındırabilir. Türkçe'de karmaşık sayılar diye adlandırılır.

complex64: Gerçel float32 ve sanal sayı değeri barındırır.
complex128: Gerçel float64 ve sanal sayı değeri barındırır.
Sayısal türler bu şekildedir.

BOOLEAN VERİ TİPİ
Boolean yani mantıksal veri tipi bir durumun var olması halinde olumlu (true) değer, var olmaması halinde olumsuz (false) değer alan veri tipidir.

STRING VERİ TİPİ
String yani dizgi veri tipi içerisinde metinsel ifadeler barındırır. Örnek olarak "Golang çok güzel ama ingilicce". String veri tipi değeri çift tırnak ("Değer") içine yazılır. Diğer dillerdeki gibi tek tırnak ('Değer') insiyatifi yoktur. Tek tırnakla kullanım başka bir amaç içindir. İlerde onu da göstereceğim.

Özet olarak Veri Tipleri
Veri tipleri atanacak değerlerimizi RAM üzerinde depolamak için kullandığımız araçlardır. Tam sayı değerler için Integer veri tiplerini, ondalık sayılar için Float veri tiplerini, mantıksal değerler için Boolean veri tipini, metinsel değerler için String veri tipini kullanırız. Karmaşık sayı değerleri için ise Complex veri tipini kullanırız.
"Türkiye" = String Tipi
1881 = Integer Tipi
10,5 = Float Tipi
True = Boolean Tipi
2+3i = Complex Tipi

Veri Tiplerinin Varsayılan Değerleri
Veri tipleri içerisine değer atanmadan oluşturulduğu zaman varsayılan bir değer alır.
Sayısal Tipler için 0,
Boolean Tipi için false,
String Tipi için "" (Boş dizgi) değeri alır.

Aritmetik Operatörler

Aritmetik operatörler programlamada matematiksel işlemler yapabilmemize olanak sağlar.

Operatör	Açıklama	Örnek
+	Toplar	2+5=7
–	Çıkarır	10-3=7
*	Çarpar	3*4=12
/	Böler	10/2=5
%	Bölümden kalanı verir(Mod).	10%3=1
++	1 arttırır	1++=2
—	1 eksiltir	3—=2

İlişkisel Operatörler

İlişkisel operatörler programlamada iki veriyi birbiriyle karşılaştırabilmemize olanak sağlar. Karşılaştırma doğrulanıyorsa **true** değer, doğrulanmıyorsa **false** değer alır.

Operatör	Açıklama	Örnek
==	İki verinin eşitse true verir	2==3 (false)
!=	İki verinin eşit değilse true verir	2!=3 (true)
>	1. veri 2. veriden büyükse true verir	5>3 (true)
<	1. veri 2. veriden küçükse true verir	4<6 (true)
>=	1. veri 2. veriden büyük veya eşitse true verir	4>=3 (true)4>=4 (true)
<=	1. veri 2. veriden küçük veya eşitse true verir	3<=2 (false)3<=3 (true)

Mantıksal Operatörler

Mantıksal operatörler birden fazla mantıksal veriyi kontrol eder ve kullandığımız operatöre göre mantıksal değer döndürür.

Operatör	Açıklama	Örnek
&&	VE operatörüdür. 2 değer de true ise true değer döndürür.	2==2 && 2==3 (false)
||	VEYA operatörüdür. 2 değerden biri true ise true değer döndürür.	2==2 || 2==3 (true)
!	DEĞİL operatörüdür.2 değer mantıksal olarak karşılaştırıldığında çıkan sonucun tersini alır.	! (2==2&&3==3) (false)

Atama Operatörleri

Atama operatörleri değişkenlere ve sabitlere değer atamak için kullanılır. Aşağıdaki tabloda c'nin değeri 10'dur. (c=10)

Operatör	Açıklama	Örnek
=	Atama operatörüdür	c=2
+=	Kendiyle toplar	c+=2(c=12
-=	Kendinden çıkarır	c-=2(c=8)
*=	Kendiyle çarpar	c*=2(c=20)
/=	Kendine böler	c/=2(c=5)
%=	Kendine bölümünden kalanı atar	c%=3(c=1)

Değişkenler ve Atanması

Değişkenler içerisinde değer barındırarak RAM'e kaydettiğimiz bilgilerdir. Değişkenler programımızın işleyişinde önemli bir role sahiptir. Değişkenleri şu şekillerde atayabiliriz. Değişkenler **var** ifadesi ile atanır. Tabi ki zorunlu değildir.

```
var isim string = "Ali"
var yas int = 20
var ogrenci boolean = true
```

Yukarıdaki yazdıklarımızı inceleyecek olursak;

var ile değişken atadığımızı belirtiyoruz. **isim** diye bir değişken adı atadık ve içine **"Ali**" değerinde **string** tipinde bir değer yerleştirdik. String tipi değerler çift tırnak içine yazılır.

Aynı şekilde **yas** adında değişken oluşturduk. **yas** değişkeni içerisine **int** tipinde **20** değerini yerleştirdik Son olarak **ogrenci** adında bir değişken oluşturduk ve **true** değerinde **boolean** tipinde bir atama yaptık.

Golang'ta değişken adı oluştururken Türkçe karakterler kullanabiliriz. Örnek olarak **ogrenci** yerine öğrenci yazabilirdik. Ama başka bir programlama diline geçtiğinizde Türkçe harf desteklememesi halinde alışkanlıklarınızı değiştirmeniz gerekecek. O yüzden Türkçe karakter kullanmamanızı tavsiye ederim. Tabi ki zorunlu değil.

Programlama dillerinde, matematiğin aksine **= (eşittir)** işareti eşitlik için değil, atamalar için kullanılır.

Değişkenlerin atanması için farklı yöntemler de var. Diğer yöntemlere değinmek gerekirse;
Değişken atamasında illaki değişkenin veri tipini belirtmemiz gerekmez. Yazdığımız değere göre Golang otomatik olarak veri tipini algılar.

```
var isim = "Ali"
var yas = 20
var ogrenci = true
```

isim değişkeninin değerini çift tırnak arasına yazdığımız için **string** veri tipinde olduğunu algılayacaktır.

yas değişkeninin değerini sayı olarak girdiğimiz için **int** tipinde olduğunu algılar. Eğer 20 değil de 2.12312 gibi küsüratlı bir değer girseydik veri tipini **float** olarak algılardı.

ogrenci değişkeninin değerini mantıksal olarak girdiğimiz için **boolean** veri tipinde olduğunu algılayacaktır.

En basit şekilde değişken ataması yapmak istersek;

```
isim:="Ali"
yas:=20
ogrenci:=true
```

Başına **var** eklemeden de değişken atamak mümkündür. Bu şekilde yapmak için **:=** işaretlerini kullanırız. Aynı şekilde bu yöntemde de verinin tipi otomatik algılanır.

Eğer değişken tanımlar iken değer kısmını boş bırakırsak yani; **var yas int** şeklinde yazarsak, önceki konuda da bahsettiğimiz gibi varsayılan olarak **0** değerini alır.

Sabitler

Sabitler de değişkenler gibi değer alır. Fakat adından da anlaşılabileceği üzere verilen değer daha sonradan değiştirilemez.

Sabitler tanımlanırken başına const eklenir. Örnek olarak;

```
const isim string = "Ali"
const isim="Veli"
```

const ile **:=** beraber kullanılamaz.

Yanlış kullanım: **const isim := "Ali"**
Doğru kullanım: **const isim = "Ali"**

Örnek olarak bir sabitin değerini atandıktan sonra değiştirmeye çalışalım. Aramızda ne olacağını merak eden çılgınlar olabilir.

Bu şekilde yazıp kodlarımızı derlediğimizde hata almamız kaçınamaz. Derlediğimizde **cannot assign to isim** hatasını verecektir. Yani diyor ki **isim'e atanamıyor**.

```
const isim string = "Ali"
isim = "Ali"
const yas = 20
```

Kod Gruplama İşlemi

Kod gruplama işlemi çok basit bir işlemdir. Bu işlem sayesinde aynı objeler bloklara göre farklı çalışabilir. Kodları gruplama için süslü parantez kullanırız. Örneğimizi görelim.

```
package main
import "fmt"
func main() {
	değişken := "bir"
	{
		değişken := "iki"
		fmt.Println(değişken)
	}
	fmt.Println(değişken)
}
```

Çıktımızı gördükten sonra kodları açıklayayım.

```
iki
bir
```

Yukarıda **değişken** isminde değişken oluşturduk. Hemen aşağısına süslü parantez oluşturduk. İçine yine değişken adında bir değişken tanımladık. Bu iki değişken aynı kod bloğunda bulunmadığı için birbirleri ile alakası olmayacaktır. Aslında ikisi de aynı değişkendir. Sadece içindeki bloğa göre farklı bir değeri vardır. Bunu anlamanın en basit yolu pointer ile bellek adresine bakmaktır. Bir o versiyonunu görelim.

```
package main
import "fmt"
func main() {
	değişken := "bir"
	{
		değişken := "iki"
		fmt.Println(değişken)
		fmt.Println(&değişken)
	}
	fmt.Println(değişken)
	fmt.Println(&değişken)
}
```

& (and) işareti ile değişkenin bellekteki adresini öğrenebiliriz.
Çıktımız şöyle olacaktır;

```
iki
0xc00008c1d0
bir
0xc00008c1c0
```

Gördüğünüz gibi bellek adresi 2 sonuçta da aynı gözüküyor. Bu ikisinin de aynı değişken olduğuna işaret ediyor.

Tür Dönüşümü

Tür dönüşümü şu şekilde gerçekleştirilir.
tür(değer)
Örnek olarak bakmak gerekir ise;

```
i := 42
f := float64(i)
u := uint(f)
```

Yukarıdaki yapılan işlemleri açıklayacak olursak eğer, i adında bir int değişken tanımladık. f adındaki değişkende i değişkenini float64 türüne dönüştürdük. u adındaki değikende ise f değişkenini uint türüne çevirdik.
Tüm türler arasında bu şekilde dönüşüm gerçekleştiremezsiniz. Bir sayıyı string tipine dönüştürmek istediğimizde ne olacağına bakalım.

```
deneme := string(8378)
fmt.Println(deneme)
```

deneme adındaki değerimizin içinde **8378** sayısını **string** türüne dönüştürdük ve hemen aşağısına **deneme**'nin aldığı değeri ekrana bastırması için kodumuzu yazdık.
Aldığımız konsol çıktısı şu şekilde olacaktır.

```
₺
```

Yani Türk Lirası simgesi çıkacaktır. Sayılar string türüne dönüştürüldüğünde karakter olarak değer alır.

Fonksiyonlar

Fonksiyonlar içlerine parametre girilebilen ve işlemler yapabilen birimlerdir. Matematikteki fonksiyonlar ile aynı mantıkta çalışan bu birimlerden bir örneği inceleyelim.

```
package main
import "fmt"
func topla(a int, b int) int {
    return a + b //a ve b'nin toplamını döndürür.
}
func main() {
    fmt.Println(topla(2, 5)) //2+5 sonucunu ekrana bastır
}
```

Yukarıdaki kodları ineleyecek olursak, foksiyonlarımızı oluşturmak için **func** anahtar kelimesini kullanırız. Yanına ise fonskiyonumuzun ismini yazarız. Parantez içine fonksiyonumuzun dışarıdan alacağı parametreler için değişken-tip tanımlaması yaparız. parantezin sağına ise fonksiyonun döndüreceği **return** değerinin tipini yazarız. Süslü parantezler içinde fonksiyonumuzun işlemleri bulunur. Son olarak return ile veri tipini belirlediğimiz değeri elde etmiş oluruz.

Main fonksiyonu içerisinde **topla(2,5)** fonksiyonu ile 2 ve 5 sayısının toplamını ekrana bastırmış olduk. Yani ekrana 7 sayısı verildi.

Fonksiyonlar istendiği kadar parametre alabildiği gibi, istenirse parametresiz de olabilir. Fonksiyonları veri return etmek yerine bir işlem yaptırmak içinde kullanabiliriz.

```
package main
import "fmt"
func yazdir() {
    fmt.Println("yazı yazdırdık")
}
func main() {
    yazdir()
}
```

yazdir adlı fonsiyonumuzun parantezine değişken tanımlamadık ve parantezin sağına fonksiyon bloğu içerisinde **return** olmadığı için veri çıkış tipini belirtmedik. Fonksiyonumuzun içerisinde sadece ekrana yazı bastırdık.

Fonksiyonlar Hakkında Ayrıntılı Bilgiler

Fonksiyon parantezi içerisine değişken tanımlanırken eğer tüm değişkenlerin türleri aynı ise sadece en sağdaki değişkenin tipini belirtmeniz yeterlidir. Örnek:

```
package main
import "fmt"
func takas(a, b string) (string, string) {
    return b, a
}
func main() {
    fmt.Println(takas("Türkiye", "Cumhuriyeti"))
}
```

Çıktımız **Cumhuriyeti Türkiye** olacaktır.
Fonksiyonlar birden fazla sonuç verebilir. Bunu yapmak için;

```
package main
import "fmt"
func islem(sayi int) (x, y int) { //return'un degiskenlerini tanımladık
    x = sayi / 2
    y = sayi * 2
    return //Burada sadece return yazıyor
}
func main() {
    fmt.Println(islem(10))
}
```

Yukarıda ise isimlendirilmiş **return** kullandık. return tipini yazdığımız paranteze bakacak olursa **(x, y int)** diyerek **return** edilecek verinin fonksiyonun blokları içerisinden çekilmesini sağladık. Böylece fonksiyon bloğununun sonundaki **return** kelimesinin yanına birşey yazmadık. Bu fonksiyonumuzun çıktısı ise **5 20** olacaktır.

Boş Tanımlayıcılar

Golang kodlarımızda bazen 2 adet değer döndüren fonksiyonlar kullanırız. Bu değerlerden hangisini kullanmak istemiyorsak, değişken adı yerine _ **(alt tire)** kullanırız.
Örneğimizi görelim:

```
package main
import "fmt"
func fonksiyonumuz(girdi int) (int, int) {
	işlem1 := girdi / 2
	işlem2 := girdi / 4
	return işlem1, işlem2
}
func main() {
	ikiyeböl, dördeböl := fonksiyonumuz(16)
	fmt.Println(ikiyeböl, dördeböl)
}
```

Gördüğünüz gibi fonksiyonumuzdan dönen iki değeri de değişkenlere atadık. Eğer birini atamak istemeseydik şöyle yapardık:

```
package main
import "fmt"
func fonksiyonumuz(girdi int) (int, int) {
	işlem1 := girdi / 2
	işlem2 := girdi / 4
	return işlem1, işlem2
}
func main() {
	ikiyeböl, _ := fonksiyonumuz(16)
	fmt.Println(ikiyeböl)
}
```

Yukarıdaki kodlarımızda fonksiyonumuzun 4'e bölme özelliğini kullanmak istemediğimizden dolayı boş tanımlama işlemi yaptık.
Boş tanımlama işlemleri çoğunlukla Golang'ta programcılar tarafından hata çıktısını kullanmak istenmediğinizde yapılıyor.

Fonksiyon Çeşitleri

Golang'ta genel olarak 3 çeşit fonksiyon yapısı bulunmaktadır. Hemen bu çeşitleri görelim

Variadic Fonksiyonlar

Variadic fonksiyon tipi ile fonksiyonumuza kaç tane değer girişi olduğunu belirtmeden istediğiniz kadar değer girebilirsiniz. Hemen örneğimize geçelim.

```
package main
import "fmt"
func main() {
   fmt.Println(toplama(3, 4, 5, 6)) //18
}
func toplama(sayilar ...int) int {
   toplam := 0
   for _, n := range sayilar {
      toplam += n
   }
   return toplam
}
          ikiyeböl, _ := fonksiyonumuz(16)
          fmt.Println(ikiyeböl)
}
```

Yukarıdaki fonksiyonumuzu inceleyelim. Vericeğimiz sayıları toplaması için aşağıda **toplama** adında bir fonksiyon oluşturduk. Fonksiyonun parametresi içerisine, yani parantezler içerisine, **sayilar** isminde **int** tipinde bir değişken tanımladık. **…** (üç nokta) ile istediğimiz kadar değer alabileceğini belirttik. **toplam** değerini mantıken doğru değer vermesi için **0** yaptık. Çünkü her sayıyı toplam değikeninin üzerine ekleyecek. **range**'in buradaki kullanım amacından bahsedeyim. **range**'i **for** döngüsü ile kullandığımızda işlem yaptığımız öğenin uzunluğuna göre işlemimizi sürdürürüz. Yani fonksiyonumuzun içine ne kadar sayı eklersek işlemimiz ona göre şekillenecektir. For ve Range işlemini daha sonraki bölümümüzde göreceğiz. **Range** kullanımında **_, n** şeklinde değişken tanımlamamızın sebebi, birinci değişken yani _, dizinin indeksini yani sıra numarasını verir. Bizim bununla bir işimiz olmadığı için _ koyarak kullanmayacağımızı belirttik. İkinci değişken ise yani **n** dizinin içindeki değeri verir yani fonksiyona girdiğimiz sayıları. Sonuç olarak bu fonksiyonda **return** ile **for** işleminden sonra tüm sayıların toplamını döndürüp **main()** fonksiyonu içerisinde ekrana bastırmış olduk.

Closure (Anonim) Fonksiyonlar

Closure fonksiyonlar ile değişkenlerimizi fonksiyon olarak tanımlayabiliriz. Örneğimize geçelim.

```
package main
import "fmt"
func main() {
   toplam := func(x, y int) int {
      return x + y
   }
   fmt.Println(toplam(2, 3))
}
```

Yukarıdaki kodlarımızı inceleyecek olursak, **main** fonksiyonunun içine **toplam** adında bir değişken oluşturduk. Bu değişkenin türünün otomatik algılanması için **:=** işaretlerimizi girdik. Değişkene değer olarak anonim bir fonksiyon (ismi olmayan fonksiyon yani) yazdık. Bu fonksiyon **x** ve **y** adında iki tane **int** değer alıyor ve **return** kısmında bu iki değeri **int** olarak döndürüyor. Aşağıdaki **Println()** fonksiyonunda ise bu değişkeni aynı bir fonksiyonmuşcasına kullandık.

Recursive (İç-içe) Fonksiyonlar

Recursive fonksiyonlar yazdığımız fonksiyonun içinde aynı fonksiyonu kullanmamız demektir. Fonksiyonumun tüm işlemler bittiğinde return olur. Örneğimize geçelim.

```
package main
import "fmt"
func main() {
   fmt.Println(faktoriyel(4))
}
```

```
func faktoriyel(a uint) uint {
    if a == 0 {
        return 1
    }
    return a * faktoriyel(a-1)
}
```

Yukarıdaki fonksiyon ile bir sayının faktöriyelini hesaplayabiliriz. Faktöriyel hakkında kısaca bir hatırlatma yapayım. Belirlediğimiz sayıya kadar olan tüm sayıların sırasıyla çarpımınına o sayının faktöriyeli denir. Yani 4 sayısının faktöriyelini bulmak istiyorsak: 1*2*3*4 işlemini yaparız. Sonuç 24'tür. Faktöriyel fonksiyonun giriş ve çıkış tiplerini uint yapmamızın sebebi ise faktöriyel sonucunu bulmak için en geriye gidildiğinde eksi değerlere geçilmemesi içindir. Ayrıca sıfırın faktöriyeli birdir. Onun için değer sıfırsa bir return etmesini istedik. Faktöriyel fonksiyonunun en alttaki return kısmında girdiğimiz sayı ile girdiğimiz sayının bir eksiğinin faktöriyelini çarpacak. Girdiğimiz sayının bir küçüğünü bulmak içinse yeniden o sayının faktöriyelini hesaplayacak. Daha sonra aynı işlemler bu sayılar içinde yapılacak. Ta ki sayı son geldiğinde yani en küçük uint değeri olan 0'a dayandığında. Daha sonra sonucu main fonksiyonu içerisinde ekrana bastırdık.

Döngüler

Programlama ile uğraşan arkadaşlarımızın da bileceği üzere, programlama dillerinde **while, do while** ve **for** döngüleri vardır. Bu döngüler ile yapacağımız işlemin belirli koşullarda tekrarlanmasını sağlayabiliriz. Golang'ta ise diğer dillerin aksine sadece **for** döngüsü vardır. Ama bu **while** ve **do while** ile yapılanları yapamayacağımız anlamına gelmiyor. Golang'taki for döngüsü ile hepsini yapabiliriz. Yani dilin yapımcıları tek döngü komutu ile hepsini yapabilmemize olanak sağlamışlar.
Gelelim for döngüsünün kullanımına. Go'da for döngüsü parametreleri parantez içine alınmaz.

STANDART FOR KULLANIMI

```
for i:=0; i <10; i++ {
   fmt.Println(i)
}
```

Açıklaması:

Döngü değişkenimiz olan **i**'ye **0** sayısal değerini verdik. **i<10** yazmamızın sebebi alt bloktaki kodun sadece **i** değeri **10** sayısal değerinden küçük olduğu zaman çalışmasını sağladık. **i++** ile ise döngü her başa sardığında **i**'ye **+1** sayı eklemesini sağladık. **for** kod bloğunun içinde ise her işlemde konsola **i**'nin değerinin bastırılmasını sağladık.
Konsol çıktımız şu şekilde olacaktır;

```
0
1
2
3
4
5
6
7
8
9
```

SADECE KOŞUL BELİRTEREK KULLANMA

Bu **for** yazım şekli while mantığı gibi çalışır. Parametrelerde sadece koşul belirtilir.

```
deger:=0
for deger <10 {
   fmt.Println(deger)
   deger++
}
```

Açıklaması:

For döngüsünden ayrı olarak **deger** adında **0** sayısal değerini alan bir değişken oluşturduk. **For** döngüsünde ise sadece koşul parametresini belirttlik. Yani döngü **deger** değişkeni **10** sayısından küçük olduğu zaman çalışacak. **For** kod bloğu içerisinde her döngü tekrarlandığında deger değişkeni ekrana basılacak ve deger değişkenine **+1** eklenecek.
Konsol çıktımız şu şekilde olacaktır;

```
0
1
2
3
4
5
6
7
8
9
```

If-Else Akışı

If ve Else kelimelerinin Türkçe karşılığına bakacak olursak;
If : Eğer, **Else :** Yoksa anlamına gelir. **If-Else** akışı koşullandırmalar için kullanılır. Diğer dillerin aksine koşul parametresi parantezler içine yazılmaz. Teorik kısmı bırakıp uygulama kısmına geçelim ki daha anlaşılır olsun

```
if koşul {
   //Koşul sağlandığında yapılacak işlemler
}else{
   //Koşul sağlanmadığında yapılacak işlemler
}
```

Yukarıdaki kod tanımına göre örnek bir program yazalım;

```
package main
import "fmt"
func main() {
   i := 5
   if i == 5 {
      fmt.Println("i'nin değeri 5'tir.")
   } else {
      fmt.Println("i'nin değeri 5 değildir.")
   }
}
```

Yukarıdaki kodları inceleyelim. i'nin değerini 5 verdik. if teriminin sağında i'nin 5 eşitliği koşulunu sorguladık. Eşitse ekrana i'nin değeri 5'tir. yazısını bastıracak. Değilse i'nin değeri 5 değildir. yazısı bastıracak. i'nin değeri 5 olduğu için ekrana i'nin değeri 5'tir. yazısını bastırdı.
If-Else akşında else kullanmamız else'nin kod bloğunu boş bırakmamız ile aynı anlama gelir.

```
i := 10
if i==10 {
   fmt.Println("i'nin değeri 10'dur.")
}
```

Yukarıda sadece **if** deyimini girdik. **Else**'yi girmedik. Burada sonuçlanan olay, **i**'nin değeri **10**'a eşitse **i**'nin değeri **10**'dur. yazısını ekrana bastırır. **Else** deyimini girmediğimiz için şartın sağlanmaması durumunda hiçbir işlem gerçekleşmez. Çıktımız **i**'nin değeri **10**'a eşit olduğu için **i**'nin değeri **10**'dur. çıkar.

ELSE-IF KULLANIMI
If-Else akışında birden fazla koşul kontrolü ekleyebiliriz. Bunu **else if** deyimi ile yapabiliriz. Kısaca bakacak olursak;

```
i := 5
if i == 5 {
   fmt.Println("i'nin değeri 5'tir.")
} else if i==3{
   fmt.Println("i'nin değeri 3'tür.")
}else{
   fmt.Println("i'nin değeri belirsiz.")
}
```

else if deyiminin yazılışını da gördük. Açıklamaya gelirsek, **else if** deyimi kendinden önceki deyimin koşulunun sağlanmaması halinde bir sonraki koşulu kontrol ettirir. **If-Else** akışında istenildiği kadar **else if** deyimi eklenebilir.

Koşullar İçerisinde Operatör Kullanımı
Koşullar içerisinden mantıksal ve ilişkisel operatörler kullanılabilir. Operatörleri görmüştük. Operatör kullanarak örnekler yapalım.

```
package main
import "fmt"
func main() {
   i := 5
   a := 3
```

```
    b := 5
    if i != a { //Birinci Koşul
        fmt.Println("i eşit değildir a")
    }
    if i == b { //İkinci Koşul
        fmt.Println("i eşittir b")
    }
    if i == b && i > a { //Üçüncü Koşul
        fmt.Println("i eşittir b ve i büyüktür a")
    }
}
```

Çıktımız şu şekilde olacaktır;

```
i eşit değildir a
i eşittir b
i eşittir b ve i büyüktür a
```

Switch Akışı

Switch kelimesinin Türkçe'deki anlamı **anahtardır**. Switch deyimi de if-else deyimi gibi koşul üzerine çalışır. Yine teorik kısmı geçip anlaşılır olması için örnek yapalım. **case** deyimi durumu ifade eder. Koşul sağlandığı zaman işleme devam edilmez.

```
package main
import "fmt"
func main() {
   i := 5
   switch i {
      case 5:
         fmt.Println("i eşittir 5")
      case 10:
         fmt.Println("i eşittir 10")
      case 15:
         fmt.Println("i eşittir 15")
   }
}
```

Çıktımız şu şekilde olacaktır;

```
i eşittir 5
```

Switch'te koşulların gerçekleşmediği zaman işlem uygulamak istiyorsak bunu default terimi ile yaparız. Örnek;

```
i := 5
switch i {
   case 5:
      fmt.Println("i eşittir 5")
   default:
      fmt.Println("i bilinmiyor")
}
```

Koşulsuz Switch

Switch'in tanımını daha iyi anlayabilmeniz için koşulsuz switch kullanımına örnek verelim. Bu yöntemde switch deyiminin yanına koşul girmek yerine case deyiminin yanına koşul giriyoruz.

```
package main
import "fmt"
func main() {
   i := 5
   switch {
      case i == 5: //i=5 olduğu için diğer case'ler sorgulanmaz
         fmt.Println("i eşittir 5")
      case i < 10:
         fmt.Println("i küçüktür 10")
      case i > 3:
         fmt.Println("i büyüktür 3")
   }
}
```

Çıktımız şu şekilde olacaktır.

```
i eşittir 5
```

Defer (Erteleme)

Defer kelimesinin Türkçe'deki karşılığı **ertelemek**tir. Bu deyimi yapacağımız işlemin başına eklersek o işlemi içerisinde bulunduğu fonksiyonun içindeki işlemlerden sonra çalıştırır. Çok karışık bir cümle kurdum ama uygulamaya geçince anlayacaksınız :)

```
package main
import "fmt"
func main() {
   defer fmt.Println("İlk Cümle")
   fmt.Println("İkinci Cümle")
}
```

Çıktımız şu şekilde olacaktır;

```
İkinci Cümle
İlk Cümle
```

Açıklamaya gelirsek ekrana **İlk Cümle** yazını bastıran satırımızın başına **defer** terimini ekledik. defer eklediğimiz satır **main()** fonksiyonunun içinde olduğu için **main()** fonksiyonundaki tüm işlemler tamamlandıktan sonra ekrana yazımızı bastırdı.
Birden fazla **defer** ekleyecek olursak;

```
package main
import "fmt"
func main() {
   defer fmt.Println("ilk Cümle")
   defer fmt.Println("İkinci Cümle")
   defer fmt.Println("Üçüncü Cümle")
   defer fmt.Println("Dördüncü Cümle")
   fmt.Println("Beşinci Cümle")
}
```

Çıktımız şu şekilde olacaktır:

```
Beşinci Cümle
Dördüncü Cümle
Üçüncü Cümle
İkinci Cümle
ilk Cümle
```

Burdan anlıyoruz ki en baştaki defer eklenen satır en son işleme tabi tutuluyor.
Hadi defer ile alakalı bir programlama alıştırması yapalım.

```
package main
import "fmt"
func main() {
   fmt.Println("Sayıyor")
   for i := 0; i < 10; i++ {
      defer fmt.Println(i)
   }
   fmt.Println("Bitti")
}
```

Çıktımız şöyle olacaktır:

```
Sayıyor
Bitti
9
8
7
6
```

```
5
4
3
2
1
0
```

Struct Metodlar

Golang'ta sınıflar yoktur. Ancak, türler üzerinden metod tanımlayabiliriz. Uzatmadan örneğimize geçelim.

```
package main
import "fmt"
type insan struct {
   isim string
   yas int
   kilo int
}
func main() {
   ali := insan{}
   ali.isim = "Ali"
   ali.yas = 20
   ali.kilo = 70
   fmt.Println(ali.isim, ali.yas, ali.kilo)
```

Şimdi biz yukarıda ne yaptık?

insan tipinde bir **struct** ürettik ve bu struct içine **isim**, **yas** ve **kilo** isminde değişkenler atadık. Böylelikle programımıza yeni bir tür kazandırdık.

main() fonksiyonumuzun içinde ise, **ali** isminde **insan** dizisi oluşturduk. Böylece **ali** isimli nesnemiz **insan** türündeki tüm özelliklerden faydalabilir oldu. Hemen aşağısında ise ali'nin **isim**, **yas** ve **kilo** değerlerini atadık.
Daha sonra ali kişisinin ismini, yaşını ve kilosunu ekrana bastırdık. Bu yönteme diğer bir tabir ile **struct metodlar** denir

Çıktımız ise şöyle olacaktır;

```
Ali 20 70
```

Struct'ın mantığını anlamamız için struct yerine başka bir tip barındıran bir örnek yapalım,

```
package main
import "fmt"
type tamsayi int
func main() {
   var sayi tamsayi = 12
   fmt.Println(sayi)
}
```

Bu sefer tipi belirlerken **struct** yerine **int** tipini yazdık. Bu demek oluyor ki içerisinde **int** gibi tamsayı değer tutabilen **tamsayi** adında bir tür oluşturduk.

main() fonksiyonumuz içerisinden de görebileceğiniz üzere aynı bir değişken ataması yapar gibi **sayi** isminde **tamsayi** tipinde içerindeki değer **12** olan bir değişken tanımladık ve bunu ekrana bastırdık. Çıktımız ise tahmin edebileceğiniz üzere **12** olacaktır.

Anonim Struct Metodlar

Golang'ta tıpkı anonim fonksiyonlar olduğu gibi anonim struct methodlar da oluşturabiliriz.
Örneğimizi görelim:

```
package main
import "fmt"
func main() {
    kişi := struct {
        ad, soyad string
    }{"Kemal", "Atatürk"}
    fmt.Println(kişi)
}
```

Yukarıda struct'ı bir değişken içerisinde tanımladık. Bunu normal struct method olarak yazmaya kalksaydık aşağıdaki gibi yazardık.

```
package main
import "fmt"
type insan struct {
    ad, soyad string
}
func main() {
    kişi := insan{"Kemal", "Atatürk"}
    fmt.Println(kişi)
}
```

Struct Metodlarda Kalıtım

Programlama dillerine aşina olan arkadaşlarımız bilir, inheritance olayı vardır. Bu olay bir class'taki verileri başka bir class'ta kullanmaya yarar. İşin garip yani Golang'ta ne inheritance vardır, ne de class. Class'a benzer struct metodlar vardır. Tabiki kalıtım yapmanın bir başka olayı var Golang'ta, struct'ımıza değişken tanıtıyorken en üste kalıtım istediğimiz struct'ı yazabiliriz.

```
type insan struct {
    boy, yas, kilo int
}
type ogrenci struct {
    insan
    sinif int
}
```

Yukarıdaki işlem ile **ogrenci** struct'ının başına **insan** ekleyerek **insan** struct'ındaki verileri almasını sağladık. Böylece kalıtım (miras) işlemini yapabildik. Yukarıdakileri örnekte kullanalım.

```
func main() {
    ali:= insan{}
    ali.boy=175
    ali.kilo=73
    ali.yas=22
    fmt.Println(ali.boy, ali.kilo, ali.yas) //175 73 22
    veli:= ogrenci{}
    veli.boy=170
    veli.yas=18
    veli.kilo=70
    veli.sinif=12
    fmt.Println(veli.boy, veli.kilo, veli.yas, veli.sinif) //170 70 18 12
}
```

Struct için Fonksiyon Oluşturma

Struct metodlar için fonksiyon oluşturmayı göstereceğim. Bu sayede Struct metodlar ile üretilen değişkenler üzerinde işlemler yapabileceğiz. Diğer dillerde class içinde belirlenmiş olan fonksiyonlara benzer bir yapıdır.
Örneğimizi görelim:

```
package main
import "fmt"
type kitap struct {
	İsim  string
	Yazar string
	Fiyat float64
}
func main() {
	bir := kitap{"Nutuk", "M. K. Atatürk", 19.99}
	bir.kitapTanıt()
}
func (k *kitap) kitapTanıt() {
	fmt.Println("İsim:", k.İsim)
	fmt.Println("Yazar:", k.Yazar)
	fmt.Println("Fiyat:", k.Fiyat, "₺")
}
```

Gelelim açıklamasına,

kitap adında bir struct oluşturduk. Bu struct'ın içerisine kitabımızın bilgileri için **İsim**, **Yazar** ve **Fiyat** adında değişkenler hazırladık. **Fiyat**'ı **float64** tipinde yapmamızın sebebi küsüratı yazabilmek için.

main() fonksiyonunun içerisinde bir isminde bir kitap oluşturduk ve yanına bilgilerini doldurduk.

Hemen altında ise **bir.kitapTanıt()** diyerek kitabımızın bilgilerini ekrana bastırdık.

Peki kitapTanıt() fonksiyonu nasıl çalışıyor?

kitapTanıt() fonksiyonunu inceleyelim. öncelikle fonksiyon isminden önce **(k *kitap)** şeklinde bir ibare görüyoruz. Bu ibare sayesinde **kitap** struct'ı ile oluşturulmuş değişkenler üzerinde fonksiyon çalıştırabiliyoruz. Fonksiyonun içerinde ise bu değişkenin bilgilerini **k** ile çekeceğimizi belirttik.

Fonksiyonumuzun içerisinde **İsim**, **Yazar** ve **Fiyat** bilgilerini ekrana bastırmak için kodlar girdik.

k.İsim dediğimizde **k** argümanı **kitap** struct'ına bağlı olduğundan nokta koyduktan sonra **İsim** değişkeni öneriler arasında belirecektir. Diğer değişkenler için de bu durum geçerlidir.
Başka kitap bilgileri ekleyip onları da ekrana **kitapTanıt**() değişkeni ile bastırabiliriz.

Örnek bir program görelim:

```
package main
import "fmt"
type hesap struct {
	Sahip string
	Nakit float64
}
func main() {
	bir := hesap{"Kaan Kuşcu", 2450.15}
	bir.paraÇek(1120.45)
}
func (h *hesap) paraÇek(miktar float64) {
	if h.Nakit >= miktar {
		h.Nakit -= miktar
		fmt.Println("Çekilen miktar:", miktar)
		fmt.Printf("%s kişisinin yeni nakti: %.2f", h.Sahip, h.Nakit)
```

```
	} else {
		fmt.Println("Bu işlem için paranız yeterli değil.")
		fmt.Println("Çekebileceğiniz en büyük miktar:", h.Nakit)
	}
}
```

Pointers (İşaretçiler)

İşaretçiler yani pointer'lar bir değerin bellekteki adresini tutar. Değişken atamalarında **& (and)** işareti değişkenin bellekteki adresini tutar. *** (yıldız)** işareti ise tutulan adresteki değeri görüntüler. Tekrardan teorik kısmı kısa tutup örneğimize geçelim.

```
package main
import "fmt"
func main() {
   i := 40
   p := &i
   fmt.Println(p) //Alacağımız benzeri çıktı: 0xc000012120
   fmt.Println(*p) //Alacağımız çıktı: 40
   *p = 35 //Dolaylı olarak i nin değerini değiştirdik
   fmt.Println(i) //Alacağımız çıktı: 35
}
```

İşaretçilerin ana görevini anlatmak gerekir ise, işaretçiler yeni bir değişken oluşturmak yerine var olan bir değişkeni işaretler ve bu değişken üzerinde işlemler yapar. Kodlar ile değişiklikler yaparak mantığını kafanızda pekiştirebilirsiniz.

Pointer İçeren Fonksiyonlar

Öncelikle hatırlatma olarak pointer konusundaki **&** ile adresi, ***** ile değeri aldığımızı unutmayalım.

Örnek olarak, bir değişkeni başka bir fonksiyon içerisinden değiştirmeye bakalım.

```
package main
import "fmt"
func onYap(s int){
    s=10
}
func main(){
    s := 5
    onYap(s)
    fmt.Println(s)
}
```

Çıktımıza baktığımızda **s**'nin değerinin **10** olmadığını görürüz. Çünkü **onYap()** fonksiyonuna verdiğimiz **s** değişkeni fonksiyon içerisine sadece kopyalandı. Yani bir gerçek (değiştirmek istediğimiz) değişken üzerinde işlem yapmadık.

İşte burada **pointer** yardımımıza koşuyor. Pointer'lar sayesinde değişkenin değeri yerine tam olarak kendisi ile işlemler yapabiliyoruz.

Yukarıdaki örneği düzgün çalışacak bir hale getirelim…

```
package main
import "fmt"
func onYap(s *int){
    *s=10
}
func main(){
    s := 5
    onYap(&s)
    fmt.Println(s)
}
```

Yukarıdaki örnekte ilk olarak **main** fonksiyonuna bakalım. **onYap**() fonksiyonuna bu sefer **&** ile **s** değişkeninin adresini gönderdik. Bu sayede direkt olarak **s** değişkeni üzerinde işlem yapacak.

onYap() fonksiyonunda ise alınacak değerin ***int** tipinde olduğunu belirttik. Yani aldığımız değişkenin ***** ile değerine erişebileceğiz. Hemen aşağısında **s** değişkenine ***s** şeklinde tanımlama yaparak, değişken yerine adrese değer tanımlıyoruz.

Çıktımız bu sefer 10 olacaktır.

Diziler (Arrays)

Diziler içlerinde bir veya birden fazla değer tutabilen birimlerdir. Bir dizideki her değer sırasıyla numaralandırılır. Numaralandırma sıfırdan başlar. Aynı şekilde örneğe geçelim.

```
package main
import "fmt"
func main() {
   var a [3]string
   a[0] = "Ayşe" //Birinci değer
   a[1] = "Fatma" //İkinci değer
   a[2] = "Hayriye" //Üçüncü değer
   fmt.Println(a) //Çıktımız: [Ayşe Fatma Hayriye]
   fmt.Println(a[1])//Çıktımız: Fatma
}
```

Gelelim kodlarımızın açıklamasına. **a** isminde içerisinde 3 tane **string** tipinde değer barındırabilen bir **dizi** oluşturduk. **a** dizisinin birinci değerine yani **0 indeksi**ne **"Ayşe"** atadık. **1** ve **2** indeksine ise **"Fatma"** ve **"Hayriye"** değerlerini atadık. **a** dizisini ekrana bastırdığımızda köşeli parantezler içinde dizinin içeriğini gördük. **a**'nın **1** indeksindeki değeri bastırdığımızda ise sadece **1** indeksindeki değeri gördük. Dizinin değerlerini tek tek olarak atayabileceğimiz gibi diziyi tanımlarken de değişkenlerini atayabiliriz.

```
package main
import "fmt"
func main() {
   a := [3]string{"Ayşe", "Fatma", "Hayriye"}
   fmt.Println(a) //Çıktımız: [Ayşe Fatma Hayriye]
}
```

Dilimler (Slices)

Dilimler bir dizideki değerlerin istediğimiz bölümünü kullanmamıza yarar. Yani diziyi bir pasta olarak düşünürsek kestiğimiz dilimi yiyoruz sadece. Örneğimize geçelim.

```
package main
import "fmt"
func main() {
   a := [6]int{2, 3, 5, 6, 7, 9}
   fmt.Println(a) //Çıktımız: [2 3 5 6 7 9]
   var b []int = a[2:4] //Dilimleme işlemi
   fmt.Println(b) //Çıktımız: [5 6]
}
```

İnceleme kısmına geçelim. **a** isminde 6 tane **int** tipinde değer alan bir dizi oluşturduk. Çıktımızın içeriğini görmek için ekrana bastırdık. Dilimleme işlemi olarak yorum yaptığım satırda ise **a** dizisinde **2** ve **4** indeksi arasındaki değerleri dizi olarak **b**'ye kaydettik. **b** dizisinin içeriğini ekrana bastırdığımızda ise dilimlenmiş alanımızı gördük. Dilimleme işleminde **[ ]** içerisine dilimlemenin başlayacağı ve biteceği indeksi yazarız.

Dilim Varsayılanları (Sıfır Değerleri)

```
package main
import "fmt"
func main() {
   a := [6]int{2, 3, 5, 6, 7, 9}
   var b []int = a[:4] //Boş bırakılan indeks 0 varsayıldı
   fmt.Println(b) //Çıktımız: [2 3 5 6]
   var c []int = a[3:] //Boş bırakıldığı için 3. index ve sonrası alındı
   fmt.Println(c) //Çıktımız: [6 7 9]
}
```

Dilim Uzunluğu ve Kapasitesi

Bir dilimin uzunluk ve kapasite değeri vardır. Dilimin uzunluğunu **len()** fonksiyonu ile, kapasitesini ise **cap()** fonksiyonu ile hesaplarız. Örneğimize geçelim.

```
package main
import "fmt"
func main() {
   a := [6]int{2, 3, 5, 6, 7, 9}
   b := a[2:4]
   fmt.Println("a uzunluk", len(a))
   fmt.Println("a kapasite", cap(a))
   fmt.Println("a'nın içeriği", a)
   fmt.Println("b uzunluk", len(b))
   fmt.Println("b kapasite", cap(b))
   fmt.Println("b'nin içeriği", b)
}
```

b dizisi ile **a** dizisini dilimlediğimiz için **b** dizisinin kapasitesi ve uzunluğu değişti. Uzunluk dizinin içindeki değerlerin sayısıdır. Kapasite ise dizinin maksimum alabileceği değer sayısıdır.

Çıktımıza bakacak olursak;

```
a uzunluk 6
a kapasite 6
a'nın içeriği [2 3 5 6 7 9]
b uzunluk 2
b kapasite 4
b'nin içeriği [5 6]
```

Boş Dilimler (Nil Slices)

Boş bir dilimin varsayılan (sıfır) değeri nil'dir. Örnek olarak;

```
package main
import "fmt"
func main() {
   var a []int
   if a == nil {
      fmt.Println("Boş")
   }
}
```

Çıktısı tahmin edeceğiniz üzere Boş yazısı olaraktır.

Make ile Dilim Oluşturma
Dilimler make fonksiyonu ile de oluşturulabilir. Dinamik büyüklükte diziler oluşturabiliriz.

```
a := make([]int, 5)
```

Burada make fonksiyonu ile uzunluğu 5 olan a adında bir dizi oluşturduk.

```
a := make([]int, 0, 5)
```

Burada ise make fonksiyonu ile uzunluğu 0, kapasitesi ise 5 olan a adında bir dizi oluşturduk.

Dilime Ekleme Yapma
Bir dilime ekleme yapmak için **append** fonksiyonu kullanılır. Hemen bir örnek ile kullanılışını görelim.

```
package main
import "fmt"
func main() {
   var a []string
   fmt.Println(a) //[ ]
   a = append(a, "Ali")
   a = append(a, "Veli")
   fmt.Println(a) //[Ali Veli]
}
```

a isminde **string** tipinde boş bir dizi oluşturduk. Hemen ardından boş olduğunu teyit etmek için **a** dizisini ekrana bastırdık. Daha sonra **a** dizisine **append** fonksiyonu ile **"Ali"** değerini ekledik. Yine aynı yöntem ile **"Veli"** değerini de ekledik. Son olarak **a** dizisinin çıktısının ekrana bastırdığımızda değerlerin eklenmiş olduğunu gördük.

```
fmt.Println(len(a), cap(a))
```

a dizisinin uzunluk ve kapasitesine baktığımızda aşağıdaki çıktıyı alırız.

```
2 2
```

Range

Range, üzerinde kullanıldığı diziyi for döngüsü ile tekrarlayabilir. Bir dilim range edildiğinde, tekrarlama başına iki değer döndürür **(return**). Birinci değer dizinin **indeks**i, ikinci değer ise bu indeksin içindeki **değer**dir. Örneğimize geçelim.

```
package main
import "fmt"
var isimler = []string{"Ali", "Veli", "Hasan", "Ahmet", "Mehmet"}
func main() {
   for a, b := range isimler {
      fmt.Printf("%d. indeks = %s\n", a, b)
   }
}
```

Yukarıdaki yazdığımız kodları açıklayalım. **isimler** isminde içerisinde **string** tipinde değerler olan bir **dizi** oluşturduk.

For döngümüz ile dizinimizdeki değerleri sıralayacak bir sistem oluşturduk. Döngümüzü açıklayacak olursak, bahsettiğimiz gibi dizi üzerinde uygulanan **range** terimi iki değer döndürecek olduğundan bu değerleri kullanabilmek için **a** ve **b** adında argüman belirledik. **range isimler** diyerek isimler dizisini kullanacağımızı belirttik. Ekrana bastırma bölümümüzde ise **%** işaretleri ile sağ taraftan hangi değerleri nerede kullanacağımızı belirttik.
Çıktımız ise şu şekilde olacaktır.

```
0. indeks = Ali
1. indeks = Veli
2. indeks = Hasan
3. indeks = Ahmet
4. indeks = Mehmet
```

Map

Map'in Türkçe karşılığında yapacağı işlemi anlatan bir çeviri olmadığı için anlamı yerine yaptığı işi bilelim. Map ile bir değişken içerisindeki dizileri bölge olarak ayırabiliriz. Çok karmaşık bir cümle oldu. O yüzden örneğimize geçelim ki anlaşılır olsun.

```
package main
import "fmt"
type insan struct {
   kisi1, kisi2, kisi3 string
}
func main() {
   var m map[string]insan
   m = make(map[string]insan)
   m["isim"] = insan{
      "Ali", "Veli", "Ahmet",
   }
   fmt.Println(m["isim"])
}
```

Yukarıda **insan** isminde bir struct metodu oluşturduk ve içerisine **string** tipinde 3 tane değişken girdik. **main()** fonksiyonumuz içerisinde ise **m** adında **map** kullanarak **string** değer saklayabilen **insan** tipinde değişken oluşturduk. **m** değişkenini **make** ile **map** dizisi haline getirdik. Hemen aşağısında ise **m** değişkenine **"isim"** adında bir bölge oluşturduk ve **insan** struct'ında belirttiğimiz gibi 3 tane **string** değer girdik. Son olarak **m** dizisinin isim bölgesindeki değerleri ekrana bastırmasını istedik. Çıktımız şöyle olacaktır;

```
{Ali Veli Ahmet}
```

Birden Fazla Bölge Ekleme

Önceki yazımızda **map** ile dizilere bölgesel hale getirmeyi gördük. Şimdi de birden fazla bölgeyi nasıl yapacağımızı göreceğiz. Örneğimize geçelim.

```
package main
import "fmt"
type insan struct {
   kisi1, kisi2, kisi3 string
}
var m = map[string]insan{
   "erkekler": insan{"Ali", "Veli", "Ahmet"},
   "kadinlar": insan{"Ayşe", "Fatma", "Hayriye"},
}
func main() {
   fmt.Println(m["erkekler"])
   fmt.Println(m["kadinlar"])
   fmt.Println(m)
}
```

Yukarıda önceki örneğimizdeki gibi **insan** struct'ı oluşturduk ve içine 3 tane **string** tipinde değer atadık. **m** adında dizi oluşturduk ve **map** ile bölgeli bir dizi olduğunu belirttik. Dizinin içerisine **"erkekler"** isminde insan tipinde bir bölge oluşturduk ve içine 3 tane **string** tipinde değerimizi girdik. Aynı işlemi **"kadinlar"** isimli bölge içinde yaptık. **main** fonksiyonumuz içerisinde **erkekler** ve **kadinlar** bölgemizi ekrana bastırdık. Son olarak **m** dizisindeki tüm içeriği ekrana bastırık.
Çıktımız ise şöyle olacaktır;

```
{Ali Veli Ahmet}
{Ayşe Fatma Hayriye}
map[erkekler:{Ali Veli Ahmet} kadinlar:{Ayşe Fatma Hayriye}]
```

Burada ayrıntıyı farkedelim. **m** dizisini ekrana bastırdığımızda **map** yeni bölgeli bir dizi olduğunu vurguluyor. **Map** ile bir bakıma dizi içerisine yeni bir dizi ekliyorsunuz. Tabi bunu **struct** metodu ile yapıyoruz.

Bölgesel Silme İşlemi

delete fonksiyonu ile silme işlemimizi yapabiliriz. Hemen örneğimize geçelim.

```
package main
import "fmt"
func main() {
    m := make(map[string]int) //m isminde string bölge isimli int değer taşıyan dizi
    m["sayi"] = 25 //sayi bölgesine 25 değerini yerleştirdik
    fmt.Println(m["sayi"]) //Çıktımız: 25
    delete(m, "sayi") //sayi bölgesindeki değeri sildik
    fmt.Println(m["sayi"]) //Çıktımız: 0 (sıfır)
}
```

Arayüz (Interface)

Arayüz (interface) sayesinde fonksiyonlardan dönen değerin tipini başka bir yerde kullanmak için şekillendirebilir. Arayüzler neslere arasındaki iletişimi sağlamak için kullanılır. Daha anlaşılır olması için örnek üzerinden inceleyelim.

```
package main
import "fmt"
type iletisim interface {
    topla() int
}
type sayilar struct {
    bir, iki int
}
func main() {
    var a iletisim
    v := sayilar{3, 4}
    a = &v
    fmt.Println(a.topla())
}
type sayi int
func (f sayi) topla() int {
    return int(f)
}
func (sayi *sayilar) topla() int {
    return sayi.bir + sayi.iki
}
```

Yazdığımız kodları inceleyelim. **iletisim** isminde **topla()** fonksiyonundan **int** değer döndüren bir **interface** (arayüz) oluşturduk. **sayilar** adında içinde **bir** ve **iki** isminde **int** tipinde değişken barındıran bir **struct** oluşturduk. **main** fonksiyonumuzda **a** isminde **iletisim** tipinde bir değişken oluşturduk. Daha sonra **v** isminde **sayilar** struct'ı için içerisinde **bir** ve **iki** değişkenine denk gelen **int** tipinde sayılarımızı girdik. İşaretçiler konumuzda gördüğümüz yöntem ile **a** değişkenini **pointer** yöntemi ile **v** değişkenine işaretledik. Ekrana bastırma kısmını açıklamak için aşağıdaki fonksiyonlara değinelim. **sayi** isminde bir **int** tür oluşturduk. Altındaki **topla()** fonksiyonu ile girilen sayıyı **int** tipine çeviren bir fonksiyon yazmış olduk. **interface** kullanmamızın ana mantığı da burada ortaya çıkıyor. **struct** ile bir tür oluşturduğumuzda içine girilen değerler **int** tipinde olsa bile dışarı çıkarılırken dizi şeklinde çıktığı için **interface** ile fonksiyonun dizi şeklinde değilde **int** tipinde veriyi kullanmasını sağlıyoruz. En son fonksiyonumuzda ise **sayilar** struct'ımızı fonksiyona dahil edip struct'ı **sayi** argümanı ile kullanacağımızı belirttik. Böylelikle **bir** ve **iki** değişkenine denk gelen sayıları toplayıp **return** edebildik. **main** fonksiyonunun son bölümünde ise ekrana toplamları bastırmış olduk. Çıktımız ise **7** olacaktır.

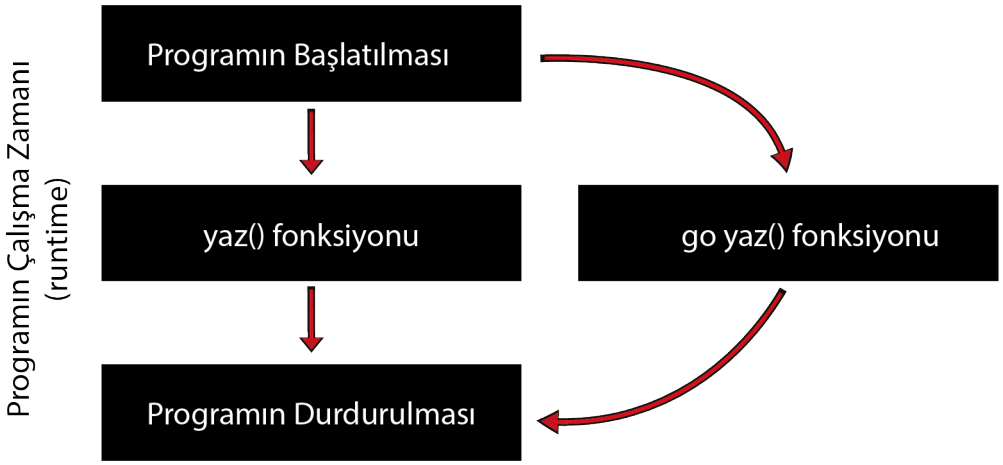
Goroutine

Goroutine'ler **Go Runtime** tarafından yönetilen hafif bir sistemdir. Bir işlemi **Goroutine** ile gerçekleştirmek istiyorsak o satırın başına **go** yazmamız yeterlidir. Şaşırtıcı ama sadece **go** yazarak bu işlemi yapabiliyoruz. Aynı **defer**'deki gibi. Goroutine aslında bir fonksiyon gibi çalışır. Eş zamanlı çalışacak fonksiyonları çağırmak için kullanılır. Basit bir örnek ile anlaşılır bir sonuç elde edelim.

```
package main
import (
   "fmt"
   "time"
)
func yaz(s string) {
   for i := 0; i < 5; i++ {
      time.Sleep(1000 * time.Millisecond)
      fmt.Println(s)
   }
}
func main() {
   go yaz("Dünya")
   yaz("Merhaba")
}
```

Yazdığımız kodları inceleyelim. Zaman ile alakalı fonksiyonları kullanabilmek için **time** paketini **import** ettik. yaz fonksiyonu oluşturduk. Bu fonksiyon **s** isminde **string** tipinde değeri işleyecek. Fonksiyonun bloğunda 5 defa 1 saniye bekleyerek istenilen yazıyı ekrana bastıran kodlarımızı girdik. **main()** fonksiyonumuzda bir tane iki tane **yaz()** fonksiyonu kullandık. Birinin başına go terimini ekledik.
Çıktımız şu şekilde olacaktır;

```
Merhaba
Dünya
Merhaba
Dünya
Dünya
Merhaba
Merhaba
Dünya
Dünya
Merhaba
```



Kanallar (Channels)

Kanallar, nesneler arasında **<-** işareti ile veri alış-verişi yapabildiğimiz hatlardır. Kanallarıda diziler gibi **make()** ile oluşturabiliriz. **make()** fonsiyonunun dinamik kapasite oluşturması sayesinden işimiz kolaylaşır. Kanallar kullanılmadan önce oluşturulmalıdır.

```
package main
import "fmt"
func topla(s []int, c chan int) {
   toplam := 0
   for _, v := range s {
      toplam += v
   }
   c <- toplam //toplam değerini c kanalına yolladık
}
func main() {
   s := []int{7, 2, 8, -9, 4, 0}
   c := make(chan int)
   go topla(s[:len(s)/2], c)
   go topla(s[len(s)/2:], c)
   x, y := <-c, <-c //c kanalından veri aldık
   fmt.Println(x, y, x+y)
}
```

topla adında bir fonksiyon oluşturduk. **s** adında **dizi** değişkeni ve **c** adında **int** tipinde bir kanal kullanacağımızı belirttik. Fonksiyonumuz içerisinde **toplam** adında **0** değerinde sayısal bir değişken oluşturduk. **Range**'i anlattığım dersten hatırlayacağınız üzere, **range**, **for** döngüsü ile bir dizi üzerine uygulandığında 2 farklı değer döndürüyordu. Burada bize **indeks** lazım olmadığından indeks yerine belirleyeceğimiz değer yerine _ **(alt tire)** kullanarak kullanıma kapattık. Gelen değerin dizi uzunluğu ile aynı kere **toplam** değişkenine eklenmesini sağladık. **for** döngümüz bittikten sonra toplam içindeki değeri **c** kanalına yolladık.

main() fonksiyonumuz içerisinde **s** adında içerisinde **int** tipinde değerler barındıran bir dizi oluşturduk. **make** fonksiyonunu kullanarak **c** adında **int** tipinde bir kanal oluşturduk. **make** ile oluşturduk ki dinamik boyutta olabilsin. **go** ile farklı dilimlemeler ile **topla** fonksiyonlarını çalıştırdık. Böylece **goroutine**'e birden fazla **thread** açtık. **c** kanalından verilerimizi alırken **x** ve **y** değişkenleri için farklı değer almış olduk. En son tüm çıktımızı ekrana bastırdık ve çıktımız bu şekilde oldu;

```
-5 17 12
```

Anonim Goroutine Fonksiyonlar

Bu yazımız **Goroutine** ve **Kanallar** dersi için biraz alıştırma tadında olacak.

Daha önceki yazılarımızda belirli bir fonksiyonu **Goroutine** ile **asenkron** (eş zamanlı) olarak çalıştırmayı gördük. Bu yazımızda da **anonim** bir **Goroutine** fonksiyonunu göreceğiz. Bu fonksiyonun özelliği bir ismi olmaması ve asenkron olarak çalışmasıdır. Örneğimizi görelim.

```
package main
import (
	"fmt"
	"time"
)
func main() {
	go func() {
		time.Sleep(time.Second * 2)
	  fmt.Println("İlk yazımız")
	}()
	fmt.Println("İkinci yazımız")
}
```

Açıklamasına gelirsek **go func()** ile anonim bir fonksiyon oluşturduk. Bu tür fonksiyonda fonksiyonumuzun sonuna **( )** parantezlerimizi yerleştirmek zorundayız. Bu fonksiyonumuz programın geri kalanı ile aynı zamanda çalışacak. Hatta programın geri kalanı ile bağlantısı bile olmayacak. Bu sebepten ötürü mantıken 2 saniye sonra işlem yapmasını belirttiğimiz için **“İkinci yazımız”** metni gözüktükten sonra **“İlk yazımız”** metni gözükeceğini tahmin etsekte **go func()** fonksiyonu yapısı gereği zaman bağımsız çalışacağı için **fmt.Println(“İkinci yazımız”)** fonksiyonu tamamlandıktan sonra **“İlk yazımız”** metni ekrana bastırılmayacaktır bile. İsterseniz programı çalıştırıp deneyebilirsiniz.

Bunun önüne geçebilmein yolu **go func()** fonksiyonundaki işlemlerin programın çalışma zamanı içerisinde sonuç vermesidir.

```
package main
import (
	"fmt"
	"time"
)
func main() {
	go func() {
		time.Sleep(time.Second * 2)
		fmt.Println("İlk yazımız")
	}()
	fmt.Println("İkinci yazımız")
	time.Sleep(time.Second * 3)
}
```

Yukarıdaki mantıkla çalışması için zamanı böyle ayarlamamız gerekir. Ama bu yöntem çok boş (gereksiz) bir yöntemdir. Her zaman böyle zamanı tahmin edemeyiz. Örnek olarak, **go func()** fonksiyonunda internet üzerinden bir dosyanın inmesini bekleyecek olsaydık tahmini bir zaman belirleyemezdik. Ki koskoca Windows bile belirleyemiyor. Çünkü bu internet hızımız ile alaklı bir şeydir. Bu yüzden garanti bir yöntem değildir.

Bundan %100 daha garantili olan yöntem **kanallar** üzerinden haberleşmektir. Çünkü bir yerde kanal ataması yapıldığında program akışının devam edebilmesi için mutlaka kanaldan gelecek verinin beklenmesi gerekir. Bu sayede zaman ile alakalı işlerde tahmin yürütmemize gerek kalmaz. Biraz uzun bir açıklama oldu ama örneğimizi görünce mantığını anlayacaksınız.

```
package main
import (
	"fmt"
	"time"
)
```

```
func main() {
	kanal := make(chan string) //kanal oluşturuyoruz
	go func() {
		time.Sleep(time.Second * 2) //2 saniye uyku
	    kanal <- "Kanal bitti" //İletişime geçiriyoruz
		fmt.Println("Anonim fonksiyon yazısı")
	}()
	fmt.Println("Öylesine bir yazı")
	fmt.Println(<-kanal) //kanaldan gelen veri bekleniyor
}
```

Öncelikle kanal ile ilgili işlemler yapabilmek için **make** fonksiyonu ile kanal oluşturduk. Hemen altında kanalımızı iletişime sokmak için öylesine bir **string** değer yolladım.

go func() fonksiyonumuz yukarıdaki örnekler ile aynıdır. Bu fonksiyonumuzun 2 saniye beklemesi olduğundan dolayı fonksiyonumuzun altındaki **"Öylesine bir yazı"** daha önce görüntülenecek. Buraya kadar ilk örnek ile aynı olayla sonuçlanıyor. Programın sonlanmasını engellemek için **<-** kanal içinden değeri bastırarak kanal iletişimini beklemesini ve bundan dolayı **"Anonim fonksiyonu yazısı"**'nı da beklemiş oluyoruz.

Anonim Goroutine fonksiyonları bu şekilde kullanabiliriz.

Print Fonksiyonu Birkaç İnceleme

Print fonksiyonu Go dilinde komut satırı üzerinde yazdırma işlemi yapmak için kullanılır. **Print** fonksiyonunun en çok kullanılan 3 çeşidine bakalım.

Print() Fonksiyonu
Bu fonksiyonun içine parametreler girerek ekrana yazdırma işlemi yapabiliriz.

```
fmt.Print("Merhaba Dünya!")
```

Çıktımız şu şekilde olacaktır:

```
Merhaba Dünya
```

Println() Fonksiyonu
Bu fonksiyon ile içine parametre girerek ekrana yazdırma işlemi yapabiliriz. Yazdırma işlemi yaptıktan sonra bir alt satıra geçer.

```
fmt.Println("satır1")
fmt.Println("satır2")
```

Çıktımız şu şekilde olacaktır:

```
satır1
satır2
```

Printf() Fonksiyonu
Gelelim işimizi görecek olan **Printf()** fonksiyonuna. Bu fonksiyon sayesinde metinsel bölümlerin arasına değişken yerleştirebiliriz.

```
dil:="Go"
yıl:=2007
fmt.Printf("%s dili %d yılından beri geliştiriliyor.",dil,yıl)
```

Çıktımız şu şekilde olacaktır;

```
Go dili 2007 yılından beri geliştiriliyor.
```

Format ve Kaçış Karakterleri

Format Karakterleri ve Kullanım Alanları

Format Karakteri	Açıklama
%T	Değişken tipini belirtir.
%t	Boolean değerini belirtir.
%d	Int değerini belirtir.
%b	Sayının binary karşılığını belirtir.
%c	Karakter değerini belirtir.
%x	Sayının hexadecimal karşılığını belirtir.
%f	Float değerini belirtir.
%s	String değerini belirtir.

Kaçış Karakterleri ve Kullanım Alanları

Kaçış Karakteri	Açıklama
\a	Komut satırında zil sesi çıkartır.
\b	Silme tuşu görevini görür.
\f	Merdiven metin yazar.
\n	Satır atlatır.
\r	Return eder.
\t	TAB tuşu gibi boşluk bırakır.
\v	Dikey boşluk bırakır.
\\	Ters-taksim yapar.
\'	Kesme işareti yapar.
\"	Tırnak işareti yapar.

Kullanıcıdan Giriş Alma

Golang'ta diğer programlama dillerinde de olduğu gibi kullanıcıdan değer girişi alınabilir. Böylece programımızı interaktif hale getirmiş oluruz.

Scan() Fonksiyonu
Bu fonksiyon boşluğa kadar olan kelimeyi kaydeder. Yeni satır boşluk olarak sayılır. Kullanımını görelim.

```
var yazi string
fmt.Scan(&yazi) //yazi değişkenine değer girilmesini istedik.
fmt.Println("\n"+yazi)
```

Yukarıda yazdığımız kodları inceleyecek olursak, belleğe **yazi** isimli **string** türünde bir değişken kaydettik. Kullanıcının girişte bulunabilmesi için **Scan()** fonksiyonunu kullandık. Bu fonksiyonun içerisine **&yazi** yazdık. Bu sayede kullanıcının girdiği değer **yazi** değişkeninin içerisine kaydedilebilecek. Daha sonra **yazi** değişkenini ekrana bastırdık ve bizim yazdığımız değer görüntülendi. **Scan** fonksiyonunda dikkat edilmesi gereken nokta kullanıcı istediği kadar kelime girse bile programın ilk kelimeyi değer olarak alacağıdır. **Scan()** fonksiyonu boş giriş kabul etmez.

Scanf() Fonksiyonu
Scanf() fonksiyonu **Printf()** fonksiyonu gibi format içerir. Bu fonksiyon ile kullanıcının girişini bölüp birkaç değişkene kaydedebiliriz. Hemen kullanımını görelim.

```
var kelime1, kelime2 string
fmt.Scanf("%s %s",&kelime1,&kelime2)
fmt.Println(kelime1)
fmt.Println(kelime2)
```

Yukarıda yazdığımız kodları inceleyecek olursak, **kelime1** ve **kelime2** adında **string** türünde değişkenler belirledik. **Scanf()** fonksiyonu ile **Printf()**'den benzer olarak, değişkenlerin yerleştirileceği yerleri değil de, bu sefer değişkenlerin alınacağı yerleri belirtiyoruz. **%s %s** arasındaki boşluk sayesinde kullanıcı boşluk bırakınca girdiyi 2 değere bölebileceğiz. Hemen yanında ise içine atanacak değişkenlerimizi belirtiyoruz. Böylelikle kullanıcı giriş bölümünden Go Dili yazdığında **Go**'yu **kelime1**'in içine **Dili** de **kelime2** içine yerleştirecek. **Scanf()**, boş giriş kabul eder.

Reader ile Satır Olarak Değer Alma
Aşağıdaki yöntem ile bir satır yazıyı giriş olarak alabilirsiniz.

```
giris := bufio.NewReader( os.Stdin)
yazi, _ := giris.ReadString('\n')
```

Strings Paketi

Strings paketi ile **string** türünde değerler üzerinde işlemler yapabiliriz. Kısaca kullanımlarından bahsedelim.

Strings.Contains() Fonksiyonu

Contains() fonksiyonu ile istediğimiz bir **string** değerin içerisinde istediğimiz bir **string** değerin olup olmadığını kontrol edebiliriz. Boolean değer verir. Eğer varsa **true** değer döndürür. Yoksa **false** değer döndürür. Ufak bir uygulama örneği yapalım.

```
package main
import (
   "fmt"
   "strings"
)
func main() {
   var eposta string
   fmt.Print("E-posta adresinizi giriniz: ")
   fmt.Scan(&eposta)
   if strings.Contains(eposta, "@") {
      fmt.Println("E-posta Adresiniz Onaylandı!")
   } else {
      fmt.Println("Geçerli Bir E-posta Adresi Giriniz!")
   }
}
```

"strings" paketini eklemeyi unutmuyoruz. Bu kodlar ile kullanıcıdan e-posta adresi isteyen ve e-posta adresi içinde @ işareti var ise olumlu yanıt veren bir programcık oluşturduk. **Constains()** fonksiyonunu açıklayacak olursak, **Contains** fonksiyonunun ilk parametresine kontrol edeceğimiz öğeyi giriyoruz. İkinci parametreye ise aranılacak **string** ifademizi giriyoruz. Gayet anlaşılır olduğunu düşünüyorum.

Strings.Count() Fonksiyonu

Count() fonksiyonu ile bir string değerin içinde istediğimiz bir string değerin kaç tane olduğunu öğrenebiliriz. Örneğimize geçelim.

```
package main
import (
   "fmt"
   "strings"
)
func main() {
   fmt.Println(strings.Count("deneme", "e"))
}
```

"strings" paketini eklemeyi unutmuyoruz. Bu kodlar ile **Count()** fonksiyonunda **"deneme"** stringi içerisinde **"e"** stringinin kaç tane geçtiğini öğreniyoruz. Çıktımız **3** olacaktır.

Strings.Index() Fonksiyonu

Index() fonksiyonu ile bir **string** değerin içindeki istediğimiz bir string değerin kaçıncı sırada yani **index**'te olduğunu öğrenebiliriz. Sıra sıfırdan başlar. Örneğimize geçelim.

```
package main
import (
   "fmt"
   "strings"
)
func main() {
   fmt.Println(strings.Index("Merhaba Dünya", "h"))
}
```

Çıktımız h harfi 0'dan başlayarak 3. sırada olduğu için, 3 olacaktır.

Strings.LastIndex() Fonksiyonu

LastIndex() fonksiyonu ile bir **string** değerin içinde istediğimiz bir **string** değerin sırasını **Index()** fonksiyonunun tersine sağdan sola doğru kontrol eder. İlk çıkan sonucun index'ini seçer. Örnek:

```
fmt.Println(strings.LastIndex("Merhaba Dünya", "a"))
```

"Merhaba Dünya" yazısının içinde **"a"** harfini aradık. **LastIndex()** fonksiyonu sondan başa yani sağdan sola arama yaptığı için sondaki **"a"** harfini buldu. Yani **13** sonucunu ekrana bastırmış olduk.

Strings.Title() Fonksiyonu

Title() fonksiyonu ile içerisine küçük harflerle **string** türünde değer girdiğimizde baş harfleri büyük harf yapan bir fonksiyondur.

```
fmt.Println(strings.Title("merhaba dünya"))
```

Çıktımız **"Merhaba Dünya"** olacaktır.

Strings.ToUpper() Fonksiyonu

ToUpper() fonksiyonu içerisine girilen string değerin tüm harflerini büyük harf yapar.

```
fmt.Println(strings.ToUpper("merhaba dünya"))
```

Çıktımız **"MERHABA DÜNYA"** olacaktır.

Strings.ToLower() Fonksiyonu

ToLower() fonksiyonu içerisine girilen string değerin tüm harflerini küçük harf yapar.

```
fmt.Println(strings.ToLower("Merhaba Dünya"))
```

Çıktımız **"merhaba dünya"** olacaktır.

Strings.ToUpperSpecial() Fonksiyonu

ToUpper() fonksiyonu ile **string** değeri büyük harf yaptığımız zaman Türkçe karakter sıkıntısı yaşarız. Örnek olarak **"i"** harfi büyüyünce **"I"** harfi olur. Bunun önüne **ToUpperSpecial()** fonksiyonu ile geçebiliriz. Bu fonksiyonun ilk parametresine karakter kodlamasını, ikinci parametresine ise **string** değerimizi gireriz. Örnek olarak:

```
fmt.Println(strings.ToUpperSpecial(unicode.TurkishCase, "ıiüöç"))
```

Çıktımız **"IİÜÖÇ"** olacaktır.

Strings.ToLowerSpecial() Fonksiyonu

ToUpperSpecial() fonksiyonu ile aynı seçilde çalışır ;fakat harfleri belirlediğiniz karakter kodlamasına göre küçültür. Örnek kullanımı:

```
fmt.Println(strings.ToLowerSpecial(unicode.TurkishCase, "IİÜÖÇ"))
```

Çıktımız **"ıiüöç"** olacaktır.

Dışa Aktarma (Exporting)

Golang'ta dışa altarma çok basit bir olaydır. Diğer programlama dillerinde **public** anahtar kelimesi olarak gördüğümüz bu olayın Golang'ta nasıl yapıldığına bakalım.
Golang'ta bunun için bir anahtar kelime yoktur.
Dışa aktarılmasını istediğimiz öğeyi oluştururken baş harfini büyük yazarız.
Örnek olarak:

```
func Topla(int x, y) int {
    return x + y
}
```

Gördüğünüz gibi **Topla()** fonksiyonunun baş harfini büyük yazdır. Peki dışa aktarma hangi durumlarda yapılır.

- Bir paket oluşturup başka bir paket içerisinden dışa aktarılan öğeyi kullanmak istiyorsak,
- Projemiz birden fazla .go dosyası içeriyorsa ve bir sayfadaki öğeyi başka sayfada da kullanmak istiyorsak,

dışa aktarma yöntemi işimizi görecektir.
Fonksiyonları dışa aktarabildiğimiz gibi değişkenleri ve sabitleride dışa aktarbiliriz. Örnek olarak:

```
var Degisken = string("değişken değerimiz")
const Sabit = string("sabit değerimiz")
```

Dışa aktarma olayı Golang'ta bu kadar basittir.

Import (Kütüphane Ekleme) Yöntemleri

1. Yöntem

```
import "fmt"
```

fmt paketini import ettik.

2. Yöntem

```
import (
    "fmt"
    "net/http"
)
```

Birden fazla paket import ettik.

3. Yöntem

```
import f "fmt"
```

fmt paketini **import** edip **f** olarak kullanacağımızı belirttik. Örnek olarak **fmt.Println()** yazmak yerine **f.Println()** yazacağız.

4. Yöntem

```
import . "fmt"
```

Dikkat ederseniz, **import** kelimesinden sonra nokta koyduk. Bu işlem sayesinde **fmt.Println()** yazmak yerine sadece **Println()** yazarak aynı işi yapmış oluruz.

5. Yöntem

```
import _ "fmt"
```

Bazen Golang yazarken kütüphaneyi ekleyip kullanmadığımız zamanlar olur. Böyle durumlarda program çalıştırılırken veya derlenirken **“eklemişsin ama kullanmamışsın”** hatası verir. import ederken _ **(alt tire)** koyarak bunun üstesinden gelebiliriz.

os/exec (Komut Satırı) Paketi Kullanımı

os/exec paketi komut satırına (cmd, powershell, terminal) komut göndermemizi sağlayan Golang ile bütünleşik gelen bir pakettir. Bu paket sayesinde oluşturacağımız programa sistem işlerini yaptırabiliriz. Örnek olarak dosya/klasör taşıma/silme/oluşturma/kopyalama gibi işlemleri yaptırabilir. Daha doğrusu komut satırı/terminal üzerinden yapabildiğimiz her işlemi yaptırabiliriz. Tabi kullandığımız işletim sistemine göre terminal komutları değiştiği için ona göre örnek vermeye çalışacağım.

Örnek 1: Komut Satırına Komut Gönderme
Ufak bir örnek ile başlayalım.

```
package main
import (
    "os"
    "os/exec"
)
func main() {
    cmd := exec.Command("mkdir", "klasörüm")
    cmd.Stdout = os.Stdout
    cmd.Run()
}
```

“mkdir klasörüm” komutu programın çalıştırıldığı dizinde **“klasörüm”** adında bir klasör oluşturur. Komut girerken dikkat etmeniz geren çok önemi bir detay var. Yazacağınız komut birden fazla kelimeden oluşuyorsa mutlaka ayrı ayrı girmelisiniz. Eğer **exec.Command()** fonksiyonuna direkt olarak **“mkdir klasörüm”** olarak girseydik, komutu tek kelime olarak algılayacaktı. Yani **string** dizisi mantığında çalışıyor bu olay. Sonuç olarak yukarıdaki gibi basit bir şekilde komut satırına komut yollayabilirsiniz.

Örnek 2: Komut Satırına Komut Gönderip Çıktısını Okuma
Yukarıda çok kolay bir şekilde komut göndermeyi gördük. Fakat iş komutun çıktısını okumaya gelince işler biraz karışıyor. Yavaştan vaziyetinizi alın :)
Aslında korkulacak bir olay yok. Yeter ki mantığını anlayalım. Şimdi yapacağımız işlemleri 4 ana parçaya bölelim.

1) Komutun tanımlanması
2) Çıktı okuyucusunun tanımlanması
3) Komutun başlatılması
4) Komutun çalışması

Hemen kodlarımıza geçelim.

```
package main
import (
    "bufio"
    "fmt"
    "os"
    "os/exec"
)
func main() {
    //komutun tanımlanması
    cmd := exec.Command("go", "version")
    cmdOkuyucu, hata := cmd.StdoutPipe()
    if hata != nil {
        fmt.Fprintln(os.Stderr, "Çıktı okunurken hata oluştu:", hata)
        os.Exit(1)
    }
    //çıktı okuyucusunun tanımlanması
    çıktı := bufio.NewScanner(cmdOkuyucu)
    go func() {
```

```
        for çıktı.Scan() {
            fmt.Println(çıktı.Text())
        }
    }()
    //komutun başlatılması
    hata = cmd.Start()
    if hata != nil {
        fmt.Fprintln(os.Stderr, "Komut başlatılamadı:", hata)
        os.Exit(1)
    }
    //komutun çalışması
    hata = cmd.Wait()
    if hata != nil {
        fmt.Fprintln(os.Stderr, "Komut çalışırken hata oluştu:", hata)
        os.Exit(1)
    }
}
```

Gelelim yukarıdaki kodların açıklamasına…
cmd adında bir değişken oluşturduk. Bu değişkenimiz sayesinde **exec.Command()** fonksiyonuyla komutlarımızı girdik. **cmd.StdoutPipe()** fonksiyonuyla gönderdiğimiz komutun çıktılarını alabiliyoruz. **cmdOkuyucu** değişkenine komut çıktımızı aldık. **hata** değişkenimize ise komut girildiğinde oluşan hata mesajını aldık.

hata değişkeninin içi boş değilse ekrana bastırmasını ve **1** numaralı çıkış kodunu vermesini istedik. Bu arada **1** numaralı çıkış kodu hatalar için kullanılır. Golang programlarında görmüyoruz ama **0** numaralı çıkış kod da işler yolunda gittiği zaman kullanılır. **C** dili kodlayan arkadaşlarımız bilir, **int main** fonksiyonunun sonuna **return 0** ibaresi girilir. Buraya kadar olan işlemlerimiz komutun tanımlanması ile ilgiliydi.

Çıktımızı okuyabilmemiz için birkaç işlem yapmamız gerekiyor. Ne yazık ki çıktımızı direkt olarak değişkene atayıp ekrana bastıramıyoruz. **çıktı** adında değişkenimizi oluşturuyoruz. Bu değişkenimiz **cmdOkuyucu** değişkenini taramaya yarayacak. Hemen aşağısında **goroutine** fonksiyonumuzda çıktı.**Scan()** döngüsü ile çıktı sonucumuzu ekrana bastırıyoruz.

Buraya kadar tanımlamalarımız yapmış bulunduk. Bundan sonra işlemlerimiz komutumuzun çalıştırılması ve sonucun beklenmesi olacak.

hata değişkenimize **cmd.Start()** fonksiyonunu atayarak komut başlatma işleminde hata oluşursa veriyi çekmesini sağladık. Hata var ise error tipindeki hata mesajımızı ekrana ve 1 numaralı hatayı ekrana bastıracak.

Son işlemimiz ise komutun sonuçlanmasının beklenmesi. hata değişkenimize **cmd.Wait()** fonksiyonunu ekleyerek bekleme işleminde oluşabilecek hatanın mesajını çekmiş olduk. Aşağısında eğer hata var ise ekrana bastırması için gerekli kodlarımızı girdik. Son olarak **1** numaralı çıkış işlemini yaptık.

Gördüğünüz gibi çıktı alma işlemi biraz daha uzun. Ama mantığını anladıktan sonra kolay bir işlem olduğunu düşüyorum.

Örnek 3: Hata Detayı Çekmeden Komut Çıktısı Alma
Eğer ben hata çıktısının detayını almak istemiyorum, benim işim sadece çıktıyla diyorsanız yapacağımız işlemler gerçekten kolaylaşıyor. Hemen kodlarımızı görelim.

```
package main
import (
    "fmt"
    "log"
    "os/exec"
)
func main() {
    cmd := exec.Command("go", "versison")
    çıktı, hata := cmd.CombinedOutput()
```

```
    if hata != nil {
        log.Fatalf("Komut hatası: %s\n", hata)
    }
    fmt.Printf(string(çıktı))
}
```

Kodlarımızın açıklamasına geçelim. **cmd** adında değişkenimizde **exec.Command()** fonksiyonu ile komutlarımızı tanımladık. **çıktı** ve **hata** değişkenimize komut çıktılarımızı aldık. Burada **hata** değişkeni sadece hata numarasını verecektir. Detayları barındırmaz. Eğer hatamız var ise ekrana bastırmasını istedik. Aşağısında ise **çıktı** değişkenimiz byte dizisi tipinde olduğu için **string**'e çevirip ekrana bastırdık.

WaitGroup ile Goroutine'in Tamamlanmasını Bekleme

Goroutine'leri Asenkron programlama yaparken kullanırız. Böylece aynı anda birden fazla işlem gerçekleştirebiliriz. Peki programımızın tüm goroutine'leri bekleme gibi bir ihtiyacı olsaydı ne yapmamız gerekirdir. Kodumuzu hemen aşağıda görelim.

```
package main
import (
    "fmt"
    "sync"
    "time"
)
func main() {
    total := 3
    // total'de kullanılan goroutine'ler için waitgroup oluştur.
    var wg sync.WaitGroup
    wg.Add(total)
    for i := 1; i <= total; i++ {
        // uzun sürecek bir goroutine ortamı oluşturuyoruz
        go uzunsürengoroutine(i, &wg)
    }
    // Tüm goroutine'lerin bitmesini bekliyoruz.
    wg.Wait()
    fmt.Println("Tamamlandı")
}
func uzunsürengoroutine(uyku int, wg *sync.WaitGroup) {
    // waitGroup tamamlanmasını garantiye alıyoruz
    defer wg.Done()
    time.Sleep(time.Duration(uyku) * time.Second)
    fmt.Println( uyku,"saniye uyku sürüyor")
}
```

Çok Satırlı String Oluşturma

Çok satırlı string oluşturmak için **(`)** işaretini kullanırız. Türkçe klavyeden **alt gr** ve **virgül** tuşuna basarak bu işareti koyabilirsiniz. İşte örnek kodumuz;

```
package main
import "fmt"
func main() {
    yazi := `Bu bir
    çok satırlı
    yazı örneğidir.
`
    fmt.Printf("%s", yazi)
}
```

Komut Satırı Bayrakları (Flags)

Komut satırı bayrakları, örnek olarak;

```
./uygulamamız -h
```

Sondaki **-h** bir flag(bayrak)'dir. Örnek bir program yazalım

```
package main
import (
	"flag"
	"fmt"
)
func main() {
	kelime := flag.String("kelime", "varsayılan kelime", "metin tipinde")
	sayi := flag.Int("sayi", 1881, "sayı tipinde")
	mantiksal := flag.Bool("mantiksal", false, "boolean tipinde")
	flag.Parse()
	fmt.Println("kelime:", *kelime)
	fmt.Println("sayi:", *sayi)
	fmt.Println("mantiksal:", *mantiksal)
}
```

Gelelim açıklamasına;
kelime isminde **string** tipinde bir **flag** oluşturduk. **flag.String()** fonksiyonu içerisinde 1. parametre komut satırından **"-kelime"** argümanıyla gireceğimizi gösteriyor. Varsayılan değeri **"varsayılan kelime"** olacak ve açıklama bölümünde **"metin tipinde"** yazacak.

sayi isminde **int** tipinde bir **flag** oluşturduk. **flag.Int()** fonksiyonu içerisinde komut satırından **"-sayi"** argümanıyla gireceğimizi belirttik. Varsayılan değeri **1881** olacak ve açıklama bölümünde **"sayı tipinde"** yazacak.

mantiksal isminde **bool** tipinde bir **flag** oluşturduk. **flag.Bool()** fonksiyonunda **"-mantiksal"** argumanıyla çağırılacağını belirttik. Varsayılan değeri **false** olacak ve açıklama bölümünde **"boolean tipinde"** yazacak.

Uygulamamızı build edelim ve ismi **uygulama** olsun.

```
go build -o ./uygulama .
```

Windows için build ediyorsanız. **./uygulama** yerine **./uygulama.exe** yazarak build edin. (Hatırlatma yapayım dedim) Build ettikten sonra örnek bir kullanımını yapalım.

```
./uygulama -kelime=Atatürk -sayi=1881 -mantiksal=true
```

Çıktımız şu şekilde olacaktır.

```
kelime: Atatürk
sayi: 1881
mantiksal: true
```

Peki bu girdiğimiz **flag** açıklamaları ne oluyor diye soracak olursanız eğer, onu da aşağıdaki komutu yazarak görebilirsiniz.

```
./uygulama -h
```

Çıktımız şu şekilde olacaktır.

```
Usage of ./uygulama:
  -kelime string
        metin tipinde (default "varsayılan kelime")
  -mantiksal
        boolean tipinde
  -sayi int
        sayı tipinde (default 1881)
```

Sort (Sıralama) Paketi Kullanımı

Golang'ta dizilerin içeriğini sıralaya bileceğimiz bütünleşik olarak gelen **"sort"** isminde bir paket mevcuttur. Bu paketin kullanımı oldukça kolaydır. Örneğimizi görelim.

```
package main
import (
	"fmt"
	"sort"
)
func main() {
	yazilar := []string{"c", "a", "b"}
	sort.Strings(yazilar)
	fmt.Println("Yazılar:", yazilar)
	sayilar := []int{7, 2, 4}
	sort.Ints(sayilar)
	fmt.Println("Sayılar:", sayilar)
	yazisirali := sort.StringsAreSorted(yazilar)
	fmt.Println("Yazılar Sıralandı mı?: ", yazisirali)
	sayisirali := sort.IntsAreSorted(sayilar)
	fmt.Println("Sayılar Sıralandı mı?:", sayisirali)
}
```

Gelelim açıklamasına;

Sıralama özelliğini kullanabilmek için **"sort"** paketini içe aktardık. **main()** fonksiyonumuzun içini inceleyelim.

yazilar isminde içerisinde rastgele harflereden oluşan bir **string** dizi oluşturduk. Hemen aşağısında **sort.Strings(yazilar)** diyerek sıralamanın **string** türünde olduğunu belirterek sıralamamızı yaptık. Altında **yazilar** değişkenimizi ekrana bastırdık.

sayilar isminde içerisinde rastgele sayılar olan **int** tipinde bir dizi oluşturduk. Hemen aşağısında **sort.Ints(sayilar)** diyerek **int** tipinde sıralamamızı yaptık. Altında **sayilar** değişkenimizi ekrana bastırdık.

Dizilerin sıralı olup olmadığını öğrenmek için de aşağıdaki işlemleri yaptık.

yazisirali değişkeninde **sort.StringsAreSorted(yazilar)** fonksiyonu ile **yazilar** dizisinin sıralı olup olmama durumuna göre **bool** değer aldık. Ve sonucu ekrana bastırdık.

sayisirali değişkeninde **sort.IntsAreSorted(sayilar)** fonksiyonu ile sayilar dizisinin sıralı olup olmama durumuna göre **bool** değer aldık. Ve sonucu ekrana bastırdık.

Yukarıdaki işlemlere göre çıktımız şu şekilde olacaktır.

```
Yazılar: [a b c]
Sayılar: [2 4 7]
Yazılar Sıralandı mı?:  true
Sayılar Sıralandı mı?: true
```

Strconv Paketi Kullanımı (string çeviri)

strconv paketi Golang ile bütünleşik gelen **string** tipi ve diğer tipler arasında çevirme işlemi yapabileceğimiz bir pakettir.
İlk olarak **"strconv"** paketimizi içe aktarıyoruz.
Aşağıda örnek kullanımılarını ve daha açıklayıcı olması için yanlarına kullanım amaçlarını yazdım.

```
package main
import (
	"fmt"
	"strconv"
)
func main() {
	//basit string-int arası çevirme
	sayi, _ := strconv.Atoi("-42") //string > int
	yazi := strconv.Itoa(-42)      //int > string
	//string'ten diğerlerine çevirme
	b, _ := strconv.ParseBool("true")       //string > bool
	f, _ := strconv.ParseFloat("3.1415", 64) //string > float
	i, _ := strconv.ParseInt("-42", 10, 64)  //string > int
	u, _ := strconv.ParseUint("42", 10, 64)  //string > uint
	//diğerlerinden string'e çevirme
	s1 := strconv.FormatBool(true)              //bool > string
	s2 := strconv.FormatFloat(3.1415, 'E', -1, 64) //float > string
	s3 := strconv.FormatInt(-42, 16)            //int > string
	s4 := strconv.FormatUint(42, 16)            //uint > string
	//Ekrana Yazdırma
	fmt.Printf("sayi: %d tip: %T\n", sayi, sayi)
	fmt.Printf("yazi: %s tip: %T\n", yazi, yazi)
	fmt.Printf("b: %t tip: %T\n", b, b)
	fmt.Printf("f: %f tip: %T\n", f, f)
	fmt.Printf("i: %d tip: %T\n", i, i)
	fmt.Printf("u: %d tip: %T\n", u, u)
	fmt.Printf("%T %T %T %T", s1, s2, s3, s4)
}
```

Çıktımız şu şekilde olacaktır.

```
sayi: -42 tip: int
yazi: -42 tip: string
b: true tip: bool
f: 3.141500 tip: float64
i: -42 tip: int64
u: 42 tip: uint64
string string string string
```

init() Fonksiyonu (Ön Yükleme)

Golang'da bir uygulama çalışırken genelde çalışan ilk fonksiyon **main()** fonksiyonu oluyor. Bazen programın açılışında ayarlamamız gereken ön durumlar oluşuyor. İşte **init()** fonksiyonu bize bu imkanı sunuyor. Ufak bir örnekle yazdıklarıma anlam katalım.

```
package main
import "fmt"
func init() {
	fmt.Println("init fonksiyonu yüklendi")
}
func main() {
	fmt.Println("main Fonksiyonu yüklendi")
}
```

Çıktımız aşağıdaki gibi olacaktır.

```
init fonksiyonu yüklendi
main Fonksiyonu yüklendi
```

Golang'taki **init()** fonksiyonunun kullanımı, farklı dillerdeki aynı işlevi gören fonksiyonlara oranla daha kolaydır. Örnek olarak **init()** fonksiyonunda veritabanı bağlantımızı, kayıt defteri işlemlerimizi veya sadece bir kez yapmamız gereken işleri yapabiliriz. Buna imkan sağlayan mantığı aşağıdaki örnekte görelim. Bu örnekte global tanımlanmış değişkenin değerini **init()** fonksiyonunda değiştirdiğimizde **main()** gibi farklı fonksiyonlarda kullanabildiğimizi göreceğiz.

```
package main
import "fmt"
var değişken string
func init() {
	değişken = "Merhaba Dünya"
}
func main() {
	fmt.Println(değişken)
}
```

Çıktımız şu şekilde olacaktır.

```
Merhaba Dünya
```

İşte **init()** fonksiyonunun böyle bir güzelliği var.

Log Paketi

Log paketi standart Golang paketleri içerisinde gelir ve programdaki olayları kaydetmemizi yarayacak bir altyapı sunar. **Log** programcının gözü kulağıdır. Bize hataları (bugs) bulmamız için kolaylık sağlar. Örneğimize geçelim.

```
package main
import (
    "log"
)
func init(){
    log.SetPrefix("KAYIT: ")
    log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
    log.Println("ön yükleme tamamlandı")
}
func main() {
    log.Println("main fonksiyonu başladı")

    log.Fatalln("ölümcül hata")

            log.Panicln("panic mesajı")
}
```

Hemen açıklamasına geçelim. İlk olarak log paketimizi içe aktarıyoruz. **init()** fonksiyonunda log paketimiz ile ilgili ön ayarları yapıyoruz.

init() fonksiyonumuzun içerisini dikkatlice inceleyelim. **log** paketimizin üzerine ayarlamalar yapıyoruz.

SetPrefix() fonksiyonu ile **log** çıktımızın satırının başında ne yazacağını belirleyebiliyoruz.

SetFlags() fonksiyonu ile **log** çıktımızın görünüşünü ayarlıyoruz. **log.Ldate** bize zamanını gösteriyor. **log.Lmicroseconds** mikrosaniyeyi ve l**og.Llongfile** ise dosya ismini ve yapılan işlem ile ilgili satırı gösteriyor.

log önyüklemizi yaptığımızı opsiyonel olarak log.**Println()** ile belirtiyoruz.

main() fonksiyonumuzun içerisini incelediğimizde ise;

log.Println() fonksiyonu ile klasik log çıktılama işlemini yapıyoruz. Fonksiyonun sonundaki **ln** bir alt satıra geçildğini gösteriyor.

log.Fatalln() fonksiyonu ile kritik hataları bildirir. **log.Println()** fonksiyonundan farkı program **1** çıkış kodu ile biter. Bu da programın hatalı bittiği anlamına gelir. Normalde sağlıklı çalışan bir program **0** çıkış kodu ile biter. **0** çıkış kodunu Golang programlama da kullanmamıza gerek kalmaz. Fakat **C** gibi dillerde ana fonksiyonun sonunda **return 0** ibaresini yazmak zorundayız.

log.Panicln() fonksiyonunda ise ekrana çıktımızı verir ve aynı zamanda bunu normal **panic()** fonksiyonu ile yapar. Çıktımız ise şöyle olacaktır.

```
KAYIT: 2019/10/10 20:29:14.107438 /home/ksc10/Desktop/deneme/main.go:10: ön yükleme tamamlandı
KAYIT: 2019/10/10 20:29:14.107529 /home/ksc10/Desktop/deneme/main.go:13: main fonksiyonu başladı
KAYIT: 2019/10/10 20:29:14.107539 /home/ksc10/Desktop/deneme/main.go:15: ölümcül hata
exit status 1
```

Gördüğünüz gibi son satırda **çıkış durumunun 1** olduğunu yazıyor.

panic mesajı programı direkt sonlandırır. **panic** mesajını daha üste yazarak deneyebilirsiniz.

XML Parsing

Golang üzerinde **XML** dosyalarını işlemeyi öğreneceğiz. Bu işlemin yapabileceğimiz hali hazırda standart Golang paketleri ile gelen **"encoding/xml"** paketi vardır. Örneğimize geçelim.
veri.xml isminde aşağıdaki gibi bir belgemiz olduğunu varsayalım.

```
<?xml version="1.0" encoding="UTF-8"?>
<üyeler>
   <üye tip="admin">
      <isim>Ahmet</isim>
      <sosyal>
         <facebook>https://facebook.com</facebook>
         <twitter>https://twitter.com</twitter>
         <youtube>https://youtube.com</youtube>
      </sosyal>
   </üye>
   <üye tip="okuyucu">
      <isim>Mehmet</isim>
      <sosyal>
         <facebook>https://facebook.com</facebook>
         <twitter>https://twitter.com</twitter>
         <youtube>https://youtube.com</youtube>
      </sosyal>
   </üye>
</üyeler>
```

XML Belgemizi Okuyalım

Bu işlemimizi yaparken **"io/ioutil"** ve **"os"** paketlerimizden faydalanacağız. Hemen kodlarımızı görelim.

```
package main
import (
        "fmt"
        "os"
)
func main() {
        // XML dosyamızı açıyoruz
        xmlDosya, err := os.Open("veri.xml")
        // Hata var mı diye kontrol ediyoruz
        if err != nil {
                fmt.Println(err)
        }
        fmt.Println("veri.xml dosyası başarıyla açıldı")
        // XML dosyamızı kapatmayı unutmuyoruz.
        defer xmlDosya.Close()
}
```

Eğer **XML** dosyası açılırken hata oluşmazsa çıktımız olumlu yönde olacaktır.

Şimde **XML** dosyasındaki verileri struct'ımıza kaydedelim. Parsing işlemi de yapacağımızdan dolayı **"encoding/xml"** paketini de içe aktarıyoruz. Hemen kodumuz geliyor.

```
package main
import (
        "encoding/xml"
        "fmt"
        "io/ioutil"
        "os"
)
```

```
type Üyeler struct {
	Alan   xml.Name `xml:"üyeler"`
	Üyeler []Üye    `xml:"üye"`
}
type Üye struct {
	Alan   xml.Name `xml:"üye"`
	Tip    string   `xml:"tip,attr"`
	İsim   string   `xml:"isim"`
	Sosyal Sosyal   `xml:"sosyal"`
}
type Sosyal struct {
	Alan     xml.Name `xml:"sosyal"`
	Facebook string   `xml:"facebook"`
	Twitter  string   `xml:"twitter"`
	Youtube  string   `xml:"youtube"`
}
func main() {
	// XML dosyamızı açıyoruz
	xmlDosya, err := os.Open("veri.xml")
	// Hata var mı diye kontrol ediyoruz
	if err != nil {
		fmt.Println(err)
	}
	// XML dosyamızı kapatmayı unutmuyoruz.
	defer xmlDosya.Close()
	//XML dosyamızı okuyoruz (byte olarak geliyor)
	byteDeğer, _ := ioutil.ReadAll(xmlDosya)
	//Yerleştirme işlemi için değişken oluşturuyoruz.
	var üyeler Üyeler
	xml.Unmarshal(byteDeğer, &üyeler)
	fmt.Println(üyeler.Üyeler)
}
```

ioutil ile Dosya Okuma ve Yazma

ioutil paketi standart Golang paketleri içerisinde gelir ve dosya işlemleri yapabilmemiz için bize fonksiyonlar sağlar.

Dosya Okuma

Hemen örneğimize geçelim. Açıklamaları kod üzerinde ilgili alanlara yazdım.

```
package main
import (
	"fmt"
	"io/ioutil"
)
// Hatayı kontrol etmek için fonksiyonumuz
func kontrol(err error) {
	if err != nil {
		panic(err)
	}
}
func main() {
	// Okunacak dosyamızı belirtiyoruz
	dosya, err := ioutil.ReadFile("dosya.txt")
	// Hata kontrolü yapıyoruz.
	kontrol(err)
	//Dosyamızın içeriğini ekrana bastırıyoruz.
	fmt.Println(string(dosya))
}
```

Dosya Yazma

```
package main
import (
	"io/ioutil"
)
// Hatayı kontrol etmek için fonksiyonumuz
func kontrol(err error) {
	if err != nil {
		panic(err)
	}
}
func main() {
	// Yazmak istediğimiz veriyi belirtiyoruz
	veri := []byte("golangtr.org")
	// Dosya yazma işlemini başlatıyoruz.
	err := ioutil.WriteFile("dosya.txt", veri, 0644) // 0644 dosya yazdırma izni oluyor.
	// Hata kontrolü yapıyoruz.
	kontrol(err)
}
```

Dosya yazdırma işleminde aynı isimde dosya varsa üzerine yazar.

ini Dosyası Okuma ve Düzenleme

ini dosyaları programımızın ayarlarını barındırabileceğimiz dosyalardır. Golang'de **ini** dosyalarını paket ekleyerek yapabiliriz. Paketimizi indirmek için aşağıdaki komutu yazıyoruz.

```
go get gopkg.in/ini.v1
```

Paketimizi indirdikten sonra ini dosyamız üzerinde işlemler yapabiliriz.
Aşağıdaki örneklerde kullanacağımız ini dosyası bu şekildedir. Dosyamızın ismi **ayarlar.ini** olsun.

```
# Yorum satırımız
uygulama_modu = geliştirme
[dizinler]
veri = ./dosyalar
[sunucu]
protokol = http
port = 8000
```

Ini Dosyası Okuma
Dosya okuma işlemimiz dizin mantığında çalışır. Örneğimizi görelim.

```
package main
import (
	"fmt"
	"gopkg.in/ini.v1"
)
func kontrol(e error) {
	if e != nil {
		panic(e)
	}
}
func main() {
	veri, err := ini.Load("ayarlar.ini")
	kontrol(err)
	fmt.Println("Uygulama Modu:", veri.Section("").Key("uygulama_modu").String())
	fmt.Println("Veri Dizini:", veri.Section("dizinler").Key("veri").String())
	fmt.Println("Bağlantı Protokolü:", veri.Section("sunucu").Key("protokol").String())
	fmt.Println("Bağlantı Portu:", veri.Section("sunucu").Key("port").MustInt(9999))
}
```

Çıktımız şu şekilde olacaktır.

```
Uygulama Modu: geliştirme
Veri Dizini: ./dosyalar
Bağlantı Protokolü: http
Bağlantı Portu: 8000
```

Ini Dosyası Düzenleme
Yine aynı **ayarlar.ini** dosyası üzerinde düzenlemeler yapalım. İşte örneğimiz:

```
package main
import (
	"gopkg.in/ini.v1"
)
func kontrol(e error) {
	if e != nil {
		panic(e)
	}
}
```

```
func main() {
	veri, err := ini.Load("ayarlar.ini")
	kontrol(err)
	// Değer atıyoruz.
	veri.Section("").Key("uygulama_modu").SetValue("ürün")
	// ini dosyamızı kaydetmeyi unutmuyoruz.
	veri.SaveTo("ayarlar.ini")
}
```

JSON Parsing

Golang ile **JSON parse** etmeye bakacağız. Hepimizin bildiği gibi günümüzde bir **API** (application programming interface) a veri göndermede ya da veri çekmede en sık kullanılan veri formatı **JSON** (javascript object notation) dur. Golang ile de kendi oluşturduğumuz verimizi (Golang struct) JSON'a dönüştürüp bir API'a request olarak gönderebilir ya da bir API'dan gelen JSON verisini Go programımızda kullanabiliriz. O halde çok uzatmadan Go programımızdaki verileri nasıl JSON'a dönüştürüz hemen bakalım...

MARSHALLING (Sıralama)

Go programında Go struct'ını JSON stringine dönüştürmek için **"encoding"** altındaki **"json"** paketini kullanıyoruz. Kullanıma ait kod örneği aşağıdaki gibidir.

```
package main
import (
	"encoding/json"
	"fmt"
	"log"
)
type kişi struct {
	isim    string
	soyisim string
	yaş     int
}
func main() {
	ali := kişi{
		isim:    "Ali",
		soyisim: "Veli",
		yaş:     20,
	}
	veri, err := json.Marshal(ali)
	if err != nil {
		log.Fatalln(err)
		return
	}
	fmt.Printf("JSON Parse Sonucu: %s", string(veri))
}
```

Şimdi de kodumuzu çalıştıralım ve sonucu görelim:

```
JSON Parse Sonucu: {}
```

Çıktımıza baktığımızda bir hata olmamasına rağmen **JSON string**'i boş görüyoruz. Yani **marshalling** başarılı olmuş gözüküyor; fakat boş bir struct'ı marshal etmiş gibi gözüküyor.

Evet durum tam da böyle. JSON marshal **sadece dışa aktarılmış (exported)** verileri marshal eder. Bildiğimiz gibi Golang'de export etmek için değişken ismi büyük harfle yazılmalıdır. İlk kodumuzda **struct** elemanlarının baş harflerini küçük yazdığımız için hiçbiri **export** edilmedi. Bu yüzden aslında boş bir struct ı marshal etmeye çalışıyoruz gibi algıladı **Json.Marshal()** fonksiyonu. Doğal olarak geriye boş bir JSON döndü. Haydi şimdi struct elemanlarının tamamını export ederek yani ilk harflerini büyük yazarak test edelim:

```
package main
import (
	"encoding/json"
	"fmt"
	"log"
)
type kişi struct {
	İsim    string
	Soyisim string
	Yaş     int
}
```

```
func main() {
	ali := kişi{
		İsim:    "Ali",
		Soyisim: "Veli",
		Yaş:     20,
	}
	veri, err := json.Marshal(ali)
	if err != nil {
		log.Fatalln(err)
		return
	}
	fmt.Printf("JSON Parse Sonucu: %s", string(veri))
}
```

Ve tekrar kodumuzu derleyelim ve sonucu görelim:

```
JSON Parse Sonucu: {"İsim":"Ali","Soyisim":"Veli","Yaş":20}
```

Görüldüğü gibi kodumuz çalıştı. Şimdi kısaca açıklayalım programımızı:

Kendi **"kişi"** tipimizi oluşturduk. Bu tipte bir örnek oluşturduk ve **ali** değişkenine atadık. Daha sonra **ali** değişkenimizi **json.Marshal()** fonksiyonu kullanarak **JSON**'a parse ettik. Bu fonksiyondan bize 2 değer dönmektedir. Bunların bir tanesi **[]byte** tipinde parse edilen verimiz, diğeri ise **error** tipinde hata durumunu gösteren mesajdır. Sonra hatayı kontrol ettik. Ve son olarak da hatalı değilse ekrana bastık. Tabii bizim verimiz **[]byte** tipindeydi, bunu daha okunur hale getirmek için **string**'e dönüştürdük.

Evet işte bu kadar. Peki diyelim ki JSON string imizi test etmek istiyoruz ve elimizde oldukça karmaşık bir string var. Bunu tek bir satırda incelemek oldukça zahmetli olabilir. İşte bu durumda imdadımıza **json.MarshalIndent()** fonksiyonu yetişiyor. Kullanımı aşağıdaki gibidir:

```
func main() {
	ali := kişi{
		İsim:    "Ali",
		Soyisim: "Veli",
		Yaş:     20,
	}
	veri, err := json.MarshalIndent(ali, "", "    ")
	if err != nil {
		log.Fatalln(err)
		return
	}
	fmt.Printf("JSON Parse Sonucu:\n%s", string(veri))
}
```

Görüldüğü gibi **JSON** için yeni bir fonksiyon kullandık. Dikkatimizi çeken bir şey fonksiyonun ek olarak 2 paremetre içermesidir. Bunlardan ilki yani 2. parametremiz **prefix** olarak geçmektedir. Yani 2. parametre her satırın başına gelmektedir. 2. si ise yani 3. parametremiz **indentation** olarak geçmektedir. Ben onu 4 boşluk olarak ayarladım. Şimdi programımızı tekrar çalıştıralım:

```
JSON Parse Sonucu:
{
    "İsim": "Ali",
    "Soyisim": "Veli",
    "Yaş": 20
}
```

Görüldüğü gibi ekrana basarken indentation ekleyerek bastı.

"encoding/json" paketi ile go struct'ımızı nasıl JSON'a parse edeceğimizi gördük. Artık **JSON** datamızı istediğimiz gibi kullanabiliriz.

Peki tam tersi olsaydı nasıl olurdu? Yani elimizde bir JSON verisi var. Bu bir sorgunun sonucu olabilir. Bunu Go struct'ımıza nasıl çevireceğiz? Çözüm: **UNMARSHALL**

UNMARSHALL

Unmarshal işlemi amaç olarak **marshal** işleminin tam tersidir. Elimizde **JSON** formatında bir veri vardır ve biz bunu Go struct'ına dönüştürmek istiyoruz. Bunun için **"encoding/json"** paketinde Unmarshal fonksiyonunu kullanırız. O halde çok uzatmadan koda bakalım:

```
package main
import (
	"encoding/json"
	"fmt"
	"log"
)
type kişi struct {
	İsim    string
	Soyisim string
	Yaş     int
}
func main() {
	jsonVeri := []byte(`{"İsim":"Latif","Soyisim":"Uluman","Yaş":23}`)
	var goVeri kişi
	err := json.Unmarshal(jsonVeri, &goVeri)
	if err != nil {
		log.Fatalln(err)
	}
	fmt.Printf("İsim - Soyisim: %s %s\nYaş: %d", goVeri.İsim, goVeri.Soyisim, goVeri.Yaş)
}
```

Evet görüldüğü gibi **string** formatındaki **JSON** verimizi önce **[]byte** formatına çevirdik sonra onu **Unmarshal** fonksiyonuna parametre olarak verdik. Sonucu da referansını verdiğimiz kişi türündeki **goVeri** değişkenine yazmak istedik. Ve **goVeri.İsim, goVeri.Soyisim** ve **goVeri.Yaş** ile bunlara erişmeye çalıştık. Bakalım sonuçlar nasıl:

```
İsim - Soyisim: Latif Uluman
Yaş: 23
```

Görüldüğü gibi Unmarshal işlemi başarılı bir şekilde gerçekleşti.

Peki bir **API**' dan gelen **JSON** verimize ait özellikleri (attribute) tam olarak bilmeseydik nasıl bir yol izlememiz gerekirdi? Yani biz burada API' dan **isim-soyisim-yas** özelliklerinin geleceğini biliyoruz; fakat bunları bilmeyebilirdik. Bu durumda unmarshal ı hangi türden bir veri tipine gerçeklememiz gerekiyor?
Çözüm: **"map"** . Evet map kullanabiliriz. Yani **key-value** (anahtar-değer) ler işimizi görür. Peki türleri ne olmalıdır. **"key"** ler için düşündüğümüzde bu **string** olacağı hepimizin aklına gelecektir. Peki Value lar ne olmalıdır? Görüldüğü gibi **isim** türü **string** iken, **yas integer** dı. O halde hepsini karşılayabilen bir veri türü olması lazım. Aklınızda bir şeyler canlanıyor mu? Evet yardımımıza **interface** yetişiyor. O halde **map** imizin türü **map[string]interface{}** olabilir.Hemen bunu da bir kod örneği ile görelim:

```
package main
import(
   "encoding/json"
   "fmt"
)
func main(){
   jsonVeri := []byte(`{"İsim":"Latif","Soyisim":"Uluman" ,"Yas":23 , "Kilo":80.25}`)
   var goVeri map[string]interface{}
   err := json.Unmarshal(jsonVeri ,&goVeri )
   if (err != nil){
      fmt.Printf("%+v" , err.Error())
      return
   }
     fmt.Printf("İsim: %+v \nSoyisim: %+v \nYas:%+v\nKilo:%+v" , goVeri["İsim"] , goVeri["Soyisim"] , goVeri["Yas"] ,
goVeri["Kilo"])
}
```

Programımızı çalıştırıp sonucu görelim:

```
İsim: Latif
Soyisim: Uluman
Yas:23
Kilo:80.25
```

Evet, görüldüğü gibi farklı türden veri tipleri olan bir **json string** ini go **map** ine dönüştürdük ve key değerleri ile de değerlere ulaştık.

Testing (Test Etme)

Hücrelerin vücumudaki yapı birimi olduğu gibi, aynı şekilde her bileşen de yazılımın birer parçasıdır. Yazılımın sağlıklı bir şekilde çalışabilmesi için, her bileşenin güvenilir bir şekilde çalışması gerekir.
Aynı şekilde vücudumuzun sağlığı hücrelerin güvenilirliği ve verimliliğine bağlı olduğu gibi, yazılımın düzgün çalışması bileşenlerin güvenilirliği ve verimliliğine bağlıdır.
Biraz biyoloji dersi gibi oldu ama sonuçta aynı mantığı yürütebiliriz.

Peki bileşenler nedir?
Yazılımın çalışması için yazılmış her bir kod parçasına denir. Bu bileşenlerin yazılımımızın sağlıklı bir şekilde çalıştırdığından emin olmamız gerekir.
Peki bu bileşenlerin sağlamlık kontrolünü nasıl gerçekleştiririz? Tabiki test ederek.
Bir test aşamsının Golang'ta nasıl göründüğünü görelim.

```
import "testing"
func TestFunc(t *testing.T){
   t.error() //testin başarısız olduğunu bildirir.
}
```

Yukarıdaki işlem Golang'ta yapılan bir birim testin temel yapısıdır. Yerleşik **testing** paketi, Golang'ın standart paketleri içerisinde gelir. Birim testi, ***testing.T** türündeki elemanı kabul eden ve bu elemanı göre hata yayınlayan bir bir işlemdir.
Bu fonksiyonların adı büyük harfle başlamalı ve birleşik olan adın devamı da bütük harfle başlamalıdır. Yani camel-case olmalıdır.

TestFunc olmalıdır ve Testfunc olmamalıdır.

Uygulama örneğimize geçelim.
Bir proje klasörü oluşturalım ve **main.go** dosyamız şöyle olsun.

```
package main
import "fmt"
func Merhaba(isim string) (çıktı string) {
	çıktı = "Merhaba " + isim
	return
}
func main() {
	selamla := Merhaba("Kaan")
	fmt.Println(selamla)
}
```

main.go dosyamızda fonksiyona adını girdiğimiz kişiyi selamlıyor. Buraya kadar gayet basit bir program. Fonksiyonlarımızı test edeceğimiz için baş harflerini büyük yazmayı unutmuyoruz. Böylelikle fonksiyonlarımızı dışarı aktarabiliriz. Test fonksiyonumuzun çalışma mantığını görmek için **main_test.go** dosyamıza bakalım.

```
package main
import "testing"
func TestMerhaba(t *testing.T) {
	if Merhaba("Kaan") != "Merhaba Kaan" {
		t.Error("Merhaba Fonksiyonunda bir sıkıntı var!")
	}
}
```

Yukarıda ise **main.go** sayfamızdaki **Merhaba** fonksiyonunu test etmek için **TestMerhaba** adında fonksiyon oluşturduk. **t *testing.T** ibaresi ile bu işlemin test etmeye yönelik bir işlem olduğunu belirttik.
Fonksiyonun içerisine baktığımızda, **Merhaba("Kaan")** işleminin sonucu **"Merhaba Kaan"** olmadığı zaman test hatası vermesini istedik. Ve gözükecek hatayı belirttik.
Test işlemi yapmak için aşağıdaki komutları komut satırına yazıyoruz.

```
go test
```

Yukarıdaki yazdığımız kodlara göre şöyle bir çıktımızın olması gerekir.

```
PASS
ok      _/home/ksc10/Desktop/deneme     0.002s
```

Eğer **TestMerhaba** fonksiyonunda test koşuluna **"Merhaba Kaan"** yerine **"Merhaba Ahmet"** yazsaydık, aşağıdaki gibi bir **go test** çıktımız olurdu.

```
--- FAIL: TestMerhaba (0.00s)
    main_test.go:7: Merhaba Fonksiyonunda bir sıkıntı var!
FAIL
exit status 1
FAIL    _/home/ksc10/Desktop/deneme     0.002s
```

Go Test Komutları

Komut	Açıklama
go test	İçerisinde bulunduğu projenin tüm test fonksiyonlarını test eder.
go test -v	Her test için ayrı bilgi verir.
go test -timeout 30s	30 saniye zaman aşımı ile test eder.
go test -run TestMerhaba	Sadece belirli bir fonksiyonu test eder.

Örnek kullanımı:
main_test.go dosyamızdaki **TestMerhaba** fonksiyonumuzu **10** saniye zaman aşımı ile test edecek komut

```
go test -timeout 30s -run TestMerhaba
```

Panic & Recover

Panic ve **Recover**, Golang'de hata ayıklama için kullanılan anahtar kelimelerdir. Size bunu daha iyi ve akılda kalıcı anlatmak için teorik anlatım yerine uygulamalı öğretim yapmak istiyorum. Böylece daha akılda kalıcı olur. Aşağıda panic durumu oluşturan bir örnek göreceğiz:

```
package main
func main() {
	sayilar := make([]int, 5)
	sayilar[6] = 10
}
```

Yukarıda **make** fonksiyonu ile sayilar adında uzunluğu **5** birimden oluşan bir **int** dizi oluşturduk. Bu bildiğimiz sayısal 5 tane değişken tutan bir dizi aslında. Ama altında sayilar dizisinin **6.** indeksine **10** değerini atamak istedik. Fakat **sayilar** dizesinin **6.** indeksi mantıken bulunmamakta. Bu haldeyken programımız **panic** hatası verecektir ve çıktımız aşağıdaki gibi olacaktır.

```
panic: runtime error: index out of range
goroutine 1 [running]:
main.main()
	/home/ksc10/Desktop/deneme/main.go:5 +0x11
exit status 2
```

Yani yukarıdaki gibi bir **panic hatası** alacağımız kesin. Panic hatasını programımızda yanlış birşeyler olduğunda ve programımızın işleyişinde hata olduğunda alırız. Panic hatası bir sıkıntı ile karşılaşıldığında alınabileceği gibi kendimizde panic hatası verdirebiliriz. Bu işlemi **panic()** fonksiyonu ile yapabiliriz. Örneğimizi görelim.

```
package main
import "fmt"
func main() {
	defer fmt.Println("Defer Yazısı")
	fmt.Println("Program başladı")
	panic("Panic Hatası oluştu")
	fmt.Println("Panikten sonraki cümle")
}
```

Yukarıdaki örneğimizi dikkatlice inceleyim. **main()** fonksiyonuna bakacak olursak, ilk satırında **defer** ile ekrana yazı bastırmasını istedik. Normal şartlarda **defer** ile yapılan işlem içerisinde bulunduğu kod bloğunun (bu örnekte main oluyor) en son çalışan işlemi olur. Daha sonra ekrana normal bir yazı bastıdık. **panic()** fonksiyonunun içini boş olarak da yazabilirdik ;ama kendi istediğimiz hatanın gözükmesini istedik. Son olarak kod bloğunun en sonunda bir cümle daha bastırdık.

Yukarıdaki kodları göre mantıken **defer** ile yapılan işlemin en son çalışması gerekirken. **panic()** fonksiyonu programın sonunu getirdiği için **panic()** fonksiyonundan önce çalışır ve **panic()** fonksiyonundan sonraki işlemler programda çalışmaz hale gelir.Hemen çıktımıza bakalım:

```
Program başladı
Defer Yazısı
panic: Panic Hatası oluştu
goroutine 1 [running]:
main.main()
	/home/ksc10/Desktop/deneme/main.go:8 +0xee
exit status 2
```

Çıktıda gördüğünüz gibi panic hatasından sonra kod bloğunun en sonundaki yazıyı bastırmak yerine panic hatasının açıklamasını bastırdı ve program sonlandı.

Bu örneğimizde Defer ile Panic arasındaki ilişkiyi öğrenmiş olduk.

Peki Recover Nedir?

Recover fonksiyonu ile **Defer** beraber çalışan bir ikilidir. **Defer** ile kullanmadan **recover**'ın bir mantığı olmaz. Bu durumu şöyle açıklayabiliriz. **Defer** ile çalışmasını istediğimiz işlemler **panic** olmasına rağmen gösterildiği için bir hata olduktan sonra **recover**'ı defer ile kullanıp **panic**'e erişebiliriz. Yani panic hatasını yakalayabiliriz. Bu anlamsız cümleleri örnek yaparak anlamlandıralım

```
package main
import (
	"fmt"
)
func main() {
	sayilar := make([]int, 5)
	defer fonksiyonumuz(sayilar)
	sayilar[6] = 10
	fmt.Println("Programın devamındaki cümle")
}
func fonksiyonumuz(sayilar []int) {
	if err := recover(); err != nil {
		fmt.Printf("Uzunluk hatası : %d\n", len(sayilar))
		fmt.Printf("Sıkıntı yok sadece bir hata oldu\n%s", err)
	}
}
```

Yukarıdaki kodumuzu inceleyecek olursak;

Aynı şekilde dizinin uzunluğundan dolayı kaynaklanan bir hata senaryosu var. **sayilar[6]**'ya değer atarken panic hatası aldığımızdan ve dolayısı ile buranın altındaki satırlar çalışmayacağınzdan dolayı **recover** işlemimizi panic hatası alınacak yerin üst kısmına **defer** ile yazmak zorundayız. **fonksiyonumuz()**'un içine bakalım. **If-Else** akışı için **err** adında değişkenimize **recover()** fonksiyonunu atadık. **Recover** fonksiyonu üzerinden eğer bir hata çıktısı gelirse diye, **err** değişkenimizin boş olup olmamasına baktık. Eğer boş değilse (yani != nil) Aşağıdaki gibi bir yazı bastırmasını ve en son olarak err değişkeni ile ekrana yazı bastırmasını istedik.

Çıktımız ise şöyle olacaktır.

```
Uzunluk hatası : 5
Sıkıntı yok sadece bir hata oldu
runtime error: index out of range
```

Burada anlatmak istediğim önemli olan nokta **Defer**, **Panic** ve **Recover** bir bütün olarak çalışmasıdır.

Select

Select ile çoklu goroutine işlemlerinin iletişimini bekleyebiliriz. Örneğimizi görelim:

```
package main
import (
	"fmt"
	"time"
)
func main() {
	k1 := make(chan string)
	k2 := make(chan string)
	go func() {
		time.Sleep(time.Second * 1)
		k1 <- "video"
	}()
	go func() {
		time.Sleep(time.Second * 3)
		k2 <- "ses"
	}()
	for i := 0; i < 2; i++ {
		select {
		case mesaj1 := <-k1:
			fmt.Println("Mesaj 1:", mesaj1)
		case mesaj2 := <-k2:
			fmt.Println("Mesaj 2:", mesaj2)
		}
	}
}
```

Yukarıdaki kodların bize ses ve video verisi sağlayacak bir programdan parça olduğu senaryosunu kuralım. Bu programda işlem yapabilmemiz için bize bu 2 verinin gelmesini beklememiz lazım. Verileri bekleme işlemini **select** ile yapıyoruz. Burada dikkat etmemiz gereken nokta 2 tane veri beklediğimiz için **for** atamalarında **i < 2** olarak girmeliyiz. Çünkü **i := 0** olduğu için **i 2** olana kadar arada 2 sayı var. Bu sayı boşluğu da 2 veri almayı beklememizi sağlıyor. Örnek olarak **i < 1** girip 2 veri almaya kalksak k2'den gelen veriyi beklemeyecek bile. Tam tersi olarak 2 veri alacağımız halde i < 4 girsek program **deadlock**'a girecektir. Yani başarısız bir program olacaktır.

Zamanlayıcılar (Tickers)

Golang'de zamanlayıcılar, belirli sürede bir tekrar etme işlemi için kullanılır. Zamanlayıcılar programın çalışma süresince veya durdurulana kadar çalışabilir. Örneğimizi görelim:

```
package main
import (
	"fmt"
	"time"
)
func main() {
	tekrar := time.NewTicker(500 * time.Millisecond) // her yarım saniyede 1
	bitti := make(chan bool)
	go func() {
		for {
			select {
			case <-bitti:
				return
			case zaman := <-tekrar.C:
				fmt.Println("Tekrar zamanı:", zaman)
			}
		}
	}()
	time.Sleep(1600 * time.Millisecond) // 1,6 saniye programı uyut
	tekrar.Stop()                      // Durdurduk
	bitti <- true                      // for döngüsünü sonlandırdık.
	fmt.Println("Tekrarlayıcı durdu!")
}
```

Açıklaması şöyledir:

tekrar adında bir zamanlayıcı oluşturduk ve bu zamanlayıcının özelliği her yarım saniyede bir tetiklenmesi.

bitti adında, boolean değer taşıyan bir kanal oluşturduk. Bu kanalın mantığı ileride anlayacaksınız.

Anonim Goroutine fonksiyonunun içine, yani **go func()**, sınırsız döngü çeviren bir **for** oluşturduk. Bu döngünün içerisinde **select** ile kanal iletişimlerimizi dinledik. Döngümüzün sonlanması için **bitti** kanalına herhangi bir veri gelmesi gerekiyor. Aşağısındaki **case**'de **zaman** değişkenimize **tekrar** zamanlayıcımız tetiklendikçe bu durum çalışacak. (**tekrar.C** ile zaman bilgisini alıyoruz.) Yani yarım saniyede bir zaman kanalına veri gelecek.

Anonim Goroutine fonksiyonu, **main()** fonksiyonundan ayrı olarak çalıştığından bu fonksiyonumuzun çalışması için ona zaman aralığı vermemiz gerekiyor. **time.Sleep(1600 * time.Millisecond)** ile **main()** fonksiyonumuzu **1,6** saniye bekletiyoruz. Bu bekleme süresi içinde **tekrar** zamanlayıcımız **3 kere** tetikleniyor. (500 * x < 1600 | x = 3) Haliyle de **3 kere** ekrana çıktımızı bastırıyor. **1,6** saniye geçtikten sonra tekrar zamanlayımızı **tekrar.Stop()** ile durduruyoruz.

bitti kanalına değer yollayarak, yukarıdaki **for** döngümüzü **return** ile sonlandırmış oluyoruz.

Ve en son ekranımıza **"Tekrarlayıcı durdu!"** yazımızı bastırıyoruz.
Çıktımız aşağıdaki gibi olacaktır:

```
Tekrar zamanı: 2019-10-15 14:08:02.002909142 +0300 +03 m=+0.500235484
Tekrar zamanı: 2019-10-15 14:08:02.502993622 +0300 +03 m=+1.000319851
Tekrar zamanı: 2019-10-15 14:08:03.002952074 +0300 +03 m=+1.500278387
Tekrarlayıcı durdu!
```

Reflection

Reflection, programın çalışma zamanında değişkenleri ve değerlerini bulma ve denetleme yeteneğidir. Yazının devamında bu olaydan detaylıca bahsediyor olacağız.

Bir değişkeni incelemek ve türünü bulmak için ne gerekir?
Çoğu zaman program yazarken değişkenin türünü ve değerini kendimiz belirtirir. Fakat aksi durumlarda yani türünü belirtmediğimizde bu işlemin nasıl olacağına bakalım. Golang tarafında bizi ilgilendiren operatör ise **:=** oluyor.

```
package main
import (
    "fmt"
)
func main() {
    i := 10
    fmt.Printf("%d %T", i, i)
}
```

Yukarıdaki ufak programımızda **i** değişkeninin türü (tipi) derlenme zamanında belirlenir. Aşağısında ise ekrana **i**'nin değerini ve tipini bastırdık.

Burada dikkat edilmesi gereken i değişkeninin **%d** ve **%T** format karakterlerinin işleyebileceği nitelikleri olmasıdır.

Seviyeyi çok az arttırarak Struct'ı işleyen bir fonksiyon yazalım. Örneğimiz:

```
package main
import (
        "fmt"
)
type kişi struct {
        kilo int
        yaş  int
        boy  int
}
func tanıt(k kişi) string {
        i := fmt.Sprintf("Kilo: %d Yaş: %d Boy: %d", k.kilo, k.yaş, k.boy)
        return i
}
func main() {
        ahmet := kişi{
                kilo: 70,
                yaş:  20,
                boy:  175,
        }
        fmt.Println(tanıt(ahmet))
}
```

Yukarıdaki kodlarımız içerisinde **tanıt()** fonksiyonuna bakacak olursak,
k isminde kişi **struct'ı** ile gelen değişkenimizi bölümlere ayırıp **string** değer çeviren **tanıt** adında bir fonksiyon oluşturduk.
main() fonksiyonunda da **ahmet** adından **kişi** struct'ında bir bir değişken oluşturup ekrana bastırdık.
Çıktımız şöyle olacaktır:

```
Kilo: 70 Yaş: 20 Boy: 175
```

Bizim bu işlemleri neden yaptığımızı birazdan anlayacaksınız. O zamana kadar benimle takılın.
Reflection işlemini yapabilmemiz için ilk olarak **reflect** paketini içeri aktarmamız gerekir.
Bir örnek daha yaparak **reflection** hakkında fikir sahibi olmaya başlayalım:

```
package main
import (
	"fmt"
	"reflect"
)
type bilgi struct {
	yaş  int
	kilo int
	boy  int
}
func bilgiAl(q interface{}) {
	tip := reflect.TypeOf(q)
	değer := reflect.ValueOf(q)
	fmt.Println("Tip:", tip)
	fmt.Println("Değer:", değer)
}
func main() {
	ahmet := bilgi{
		yaş:  22,
		kilo: 69,
		boy:  173,
	}
	bilgiAl(ahmet)
}
```

Burada dikkat edilmesi gereken nokta struct'ımızın yansımasını almak için **interface**'den faydalandık. **reflect.TypeOf()** fonksiyonu ile tipini, **reflect.ValueOf()** ile de içerisindeki değerleri yakaladık. **Reflection** ile ilgili biraz fikrimiz oluşmaya başladığına göre birkaç fonksiyona daha bakabiliriz. Örneğimizi görelim:

```
func bilgiAl(q interface{}) {
	tip := reflect.TypeOf(q)
	tür := tip.Kind()
	fmt.Println("Tip:", tip)
	fmt.Println("Tür:", tür)
}
```

tip.Kind() diyerek tip değikeninin türünü öğrendik. Burada struct değerini bastıracaktır.

```
func bilgiAl(q interface{}) {
	tip := reflect.TypeOf(q)
	aSayi := tip.NumField()
	fmt.Println("Tip:", tip)
	fmt.Println("Alan Sayısı:", aSayi)
}
```

tip.NumField() ile struct'ımızdaki alan sayısını öğrenebiliriz.
Peki bu reflection olayının format karakterleri ile bağlatısı nedir diyecek olursak, bir örnek daha görelim

```
package main
import (
	"fmt"
	"reflect"
)
func main() {
	a := 56
	x := reflect.ValueOf(a).Int()
	fmt.Printf("Tip:%T Değer:%v\n", x, x)
```

```
	b := "Merhaba"
	y := reflect.ValueOf(b).String()
	fmt.Printf("Tip:%T Değer:%v\n", y, y)
}
```

Yukarıdaki örneğe baktığımıda reflection işlemi ile istediğimiz değişkene istediğimiz nesnenin özelliklerini yansıtabiliyoruz.
Kısacası reflect paketi bunun için kullanılır.

net/http ile Web Server Oluşturma

Golang'ta web sunucusu oluşturma çok basit bir işlemdir.
İlk örneğimizde **localhost:5555** üzerinde çalışacak olan bir web sunucusu oluşturacağız.

```
package main
import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Merhaba %s", r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":5555", nil)

    fmt.Println("Web Sunucu")
}
```

Tarayıcınız üzerinden **localhost:5555**'e girdiğinizde sayfada sadece **Merhaba** yazdığını göreceksiniz. Daha sonra adrese **/ahmet** yazıp girdiğiniz zaman yazının **Merhaba ahmet** olarak değiştiğini göreceksiniz.

Peki bu olayın açıklaması nedir?
main() fonksiyonunun içerisinde 2 temel fonksiyon bulunuyor. **HandleFunc()** fonksiyonu belirlediğimiz adrese girildiğinde hangi fonksiyonun çalıştırılacağınız belirliyor. **ListenAndServe()** fonksiyonu ise sunucunun ayağa kalkmasını ve istediğimiz bir porttan ulaşılmasını sağlıyor.
Eğer sunucuya dosya verme yoluyla işlem yapmasını istiyorsak aşağıdaki yönteme başvurmalıyız.
index.html adında bir dosya oluşturuyoruz. İçine aşağıdakileri yazıyoruz ve kaydediyoruz.

```
<!DOCTYPE html>
<html lang="tr">
<head>
    <title>Sayfa Başlığı</title>
</head>
<body>
    Merhaba Dünya
</body>
</html>
```

Şimde de sunucu işlemlerini gerçekleştireceğimiz **main.go** dosyamızı oluşturalım.

```
package main
import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func loadFile(fileName string) (string, error) {
    bytes, err := ioutil.ReadFile(fileName)
    if err != nil {
        return "", err
    }
    return string(bytes), nil
}
```

```
func handler(w http.ResponseWriter, r *http.Request) {
    var body, _ = loadFile("index.html")
    fmt.Fprintf(w, body)
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":5555", nil)
}
```

Tarayıcıdan **localhost:5555** adresine girdiğimiz zaman oluşturmuş olduğumuz **index.html** dosyasının görüntülendiğini göreceksiniz.
Açıklayacak olursak eğer;

loadFile() fonksiyonumuz **index.html** programa aktarıldığında byte türünde olduğu için onu okuyabileceğimiz **string** türüne çevirdi. Bu özellik programımıza **“io/ioutil”** paketi sayesinde eklendi. Geri kalan kısımdan zaten yukarıda bahsetmiştik.

Statik Kütüphanesi ile Dosyaları Uygulamaya Gömme

Golang'ın müthiş yanlarından biri de bir uygulamayı build ettiğimizde bize tek çalıştırılabilir dosya şeklinde sunmasıdır. Fakat bu özellik sadece **.go** dosyalarını birleştirilmesinde kullanılıyor. Harici dosyalar programa gömülmüyor. Fakat **Statik** isimli kütüphane ile bu işlem mümkün kılınıyor.

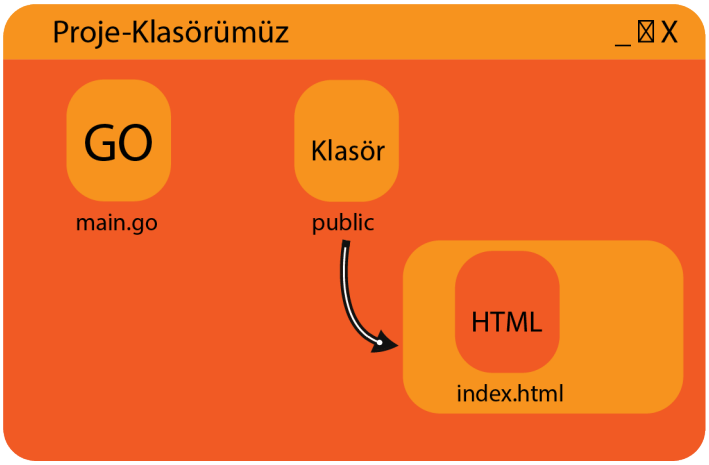
Kütüphanenin mantığından kısaca bahsedeyim. Belirlediğiniz bir dizindeki dosyaları bir kodlamaya çevirerek programın içine dosya gömmek yerine kod gömüyor. Ve bu kodu sanki dosyaymışcasına kullanabiliyoruz. Tabi ki sadece sunucu işlemlerinde işe yarar olduğunu belirtelim.
Bu yöntemin güzel artı yönleri var.

- Programımız tek dosya halinde kalıyor.
- Programımız kapalı kaynak oluyor.

Tanıtımını yaptığımıza göre hafiften uygulamaya başlayalım.

```
go get github.com/rakyll/statik
```

Konsola yukarıdakini yazarak kütüphanemizi indiriyoruz.
Öncelikle dosya ve klasör yapımızı aşağıdaki gibi ayarlıyoruz.



Kodlamaya dönüştürülmesini istediğimiz klasör ile işlem yapıyoruz. Yani **public** klasörü ile. Aşağıdaki komutu **Proje klasörümüz** içerisindeyken yazıyoruz.

```
statik -src=/public/klasörünün/adresi -f
```

Bu işlemle birlikte **public** klasörümüzün yanına **statik** isimli bir klasör oluşturduk ve içine **statik.go** isimli dosya oluşturmuş olduk. Bu dosyanın içerisinde bizim **public** klasörümüzün kodlanmış hali mevcuttur.

Ve sırada **main.go** dosyamızı oluşturmakta. Aşağıdaki kodlarıda **main.go** dosyamıza yazıyoruz.

```
package main
import (
    "net/http"
    "github.com/rakyll/statik/fs"
    _ "./statik" //Oluşturulmuş statik.go dosyasının konumu
)
func main() {
    statikFS, _ := fs.New()
    http.Handle("/", http.StripPrefix("/", http.FileServer(statikFS)))
    http.ListenAndServe(":5555", nil)
}
```

Gerekli kütüphanelerimizi ekledikten sonra main() fonksiyomuzun içeriğini inceleyelim.

statikFS ve _ adında değişkenlerimizi tanımladık. Bu değişkenlerimizi opsiyonel değişkendir. _ koymamızın sebebi **error** çıktısını kullanmak istemediğimizdendir. Eğer lazım olursa kullanabilirsiniz. **fs.New()** diyerek **statikFS** değişkenimizi bir dosya sistemi olarak tanıttık. Daha sonra sunucu oluşturak anadizine ulaşılmak istendiğinde oluşturduğumuz dosya sistemine bağlanmasını sağladık. Artık dosya sistemimize **localhost:5555** üzerinden ulaşılabilir oldu.

HTML Şablonlar (Templates)

Golang'ta **HTML** sayfalarına öğe yerleştirmek için şabloblar kullanılır. Bu işlemin uygulanışı, **.html** dosyamızın içinde Golang tarafından gelecek öğeler için işaret bırakırız. Bu işaret bu şekilde olur: **{{ kodumuz }}**

Hemen bir örnek ile olayı anlayalım. **main.go** dosyamız şöyle olsun.

```
package main
import (
	"fmt"
	"html/template"
	"net/http"
)
// SayfaBilgi ...
type SayfaBilgi struct {
	Başlık string
	İçerik string
}
func anasayfa(w http.ResponseWriter, r *http.Request) {
	sayfa := SayfaBilgi{Başlık: "Golang Türkiye", İçerik: "Sitemize Hoşgeldiniz"}
	şablon, _ := template.ParseFiles("sablonumuz.html")
	şablon.Execute(w, sayfa)
}
func main() {
	fmt.Println("Program Başladı")
	http.HandleFunc("/", anasayfa)
	http.ListenAndServe(":8000", nil)
}
```

İlk olarak sunucu oluşturacağımız için **"net/html"** ve şablon oluşturacağımız için de **"html/template"** paketlerini içe aktarıyoruz.

SayfaBilgi adında bir **struct** metod oluşturuyoruz ve içerisine **string** değer alan **Başlık** ve **İçerik** türünü oluşturuyoruz. Bunu yapmamızın sebebi web sayfamızda sayfamızın başlığını ve içeriğini bunlar aracılığıyla şablona göndereceğiz.
anasayfa adında fonksiyonumuzun içerisini inceleyelim. Bu fonksiyonumuz bir sayfa yakalayıcı fonksiyondur.

sayfa adında değişken oluşturuyoruz ve bu değişkenin **SayfaBilgi** strunct'ından olduğunu belirtip içerisine sayfa bilgilerimizi giriyoruz.

şablon değişkeni oluşturduk. (_ alt tire yerine hata bilgilerini alan değişken koyabilirsiniz.) **template.ParseFiles()** fonksiyonu ile HTML şablonumuzu tanıttık. Hemen altında şablonumuzu çalıştırması için **Execute()** fonksiyonundan yaralandık. **main()** fonksiyonumda ise klasik bir server ayağa kaldırma kodları yer alıyor.

Şimdi de **sablonumuz.html** dosyasını görelim.

```
<h1>{{.Başlık}}</h1>
{{.İçerik}}
```

Süslü parantezler içerisin **nokta** ile başlayan değişken yerleştirmelerini yapıyoruz. Bu değişkenler bize **SayfaBilgi** struct'ından gelmektedir.

Seviyeyi biraz daha yükseltelim ve listeleme işlemi yapalım.
main.go dosyamızı aşağıdaki gibi oluşturalım.

```
package main
import (
	"fmt"
	"html/template"
	"net/http"
)
```

```
// Görev ...
type Görev struct {
	İsim       string
	Tamamlandı bool
}
// SayfaVerisi ...
type SayfaVerisi struct {
	SayfaİsmiGörevListesi []Görev
}
func anasayfa(w http.ResponseWriter, r *http.Request) {
	sayfa := SayfaVerisi{
		SayfaİsmiGörevListesi: []Görev{
			{İsim: "Ekmek Al", Tamamlandı: false},
			{İsim: "Kola Al", Tamamlandı: true},
			{İsim: "Yoğurt Al", Tamamlandı: false},
		},
	}
	şablon, _ := template.ParseFiles("sablonumuz.html")
	şablon.Execute(w, sayfa)
}
func main() {
	fmt.Println("Program Başladı")
	http.HandleFunc("/", anasayfa)
	http.ListenAndServe(":8000", nil)
}
```

Bu sefer farkettiyseniz 2 tane struct metodumuz var. **Görev** struct'ımız içerisinde 2 tane değişkene ev sahipliği yapacak. **SayfaVerisi** struct'ında ise **SayfaİsmiGörevListesi** adında elemanlar var. **GörevListesi** elemanı **Görev** struct'ı türündedir. Bu sayede içerisine **dizi** olarak görevler kaydedebileceğiz.

Bir önceki örnekteki gibi **anasayfa** yakalayıcı fonksiyonumuzu oluşturuyoruz. İçerisinde **sayfa** isminde değişken oluşturuyoruz. Bu değişken içerisine sayfamızda görünmesini istediğimiz **SayfaİsmiGörevListesi** elemanlarını giriyoruz. Hemen aşağısında ise şablonumuzu bağlama işlemlerini yapıyoruz.
main() fonksiyonumuz ise bir önceki örnek ile aynıdır.
Şimdide sablonumuz.html dosyasını görelim.

```
<style>
.kirmizi{
    color:red;
}
.yesil{
    color:green;
}
</style>
<h1>{{.SayfaİsmiGörevListesi}}</h1>
<ul>
    {{range .GörevListesi}}
        {{if .Tamamlandı}}
            <li class="yesil">{{.İsim}}</li>
        {{else}}
            <li class="kirmizi">{{.İsim}}</li>
        {{end}}
    {{end}}
</ul>
```

Yukarıdaki kodları incelediğinizde içerisinde **range**, **if**, **else** ve **end** gibi kelimeler göreceksiniz. **range** anahtar kelimesi Golang'taki gibi belirtilen dizinin uzunluğu kadar sıralama işlemi yapar. **{{range}}** anahtar kelimesi mutlaka **{{end}}** ile kapatılmalıdır. **if-else** akışının ne işe yaradığını zaten biliyorsunuzdur. Aynı şekilde **{{end}}** ile kapatılmalıdır.

Tarayıcımızda **http://localhost:8000** adresini açıyoruz. Sonucu görüyoruz..

İşte Kullanabileceğimiz bazı şablon kodları:

Şablon Kodu	Açıklama
{{/* yorum */}}	Yorum yapmak için kullanılır.
{{.}}	Golang'tan ana değişken için kullanılır.
{{.DeğişkenİsmI}}	Golang'tan belirli bir değişken almak için kullanılır.
{{if .Tamamlandı}} {{else}} {{end}}	If-Else akışı için kullanılır.
{{block "içerik" .}} {{end}}	İçerik ismine bloklama tanımlar.
{{range .Görevler}} {{.}} {{end}}	Dizi sıralamak için kullanılır.

Veritabanı

Veritabanları verilerimizi kaydedip üzerinde işemler yapabildiğimiz tablosal mantıkla çalışan yapılardır. Kaydedilmesiniz istediğimiz verileri veritabanları sayesinde uzak sunucu veya yerel depolama ile kaydedip programımızın bu verilere göre işlemler yapmasını sağlayabiliriz. Bu sayede hafızası olan bir program yapmış oluruz ve program her açılışta baştan başlaması yerine kaydettiğimiz veriler ile özelleştirilebilir bir hal almış olur.

Golang'tr veritabanı işlemlerini kütüphaneler ile yapabiliriz. Programımızın işlevine göre veritabanı kütüphanesi seçimini yapabiliriz. Örneğin verilerimizi bir sunucu üzerinden alacaksak **MySQL**, **MongoDB** gibi veri tabanı sürücü kütüphanelerini kullanabiliriz. Fakat veriler bilgisayarda saklansın ve internete gerek duymasın diyorsak, yerel depolama ile **SQLite** gibi veri tabanı sürücülerini kullanabiliriz.

Golang'ta veritabanı işlemleri yapmak oldukça kolaydır. Zaten çoğu dilde veritabanı işlemleri için belirli kod kalıpları vardır. Veritabanları satır ve sütunlardan oluşur. Üzerinde tabloyu ve sütunları belirterek işlemler yapabiliriz.

sqlite3 Kütüphanesi

sqlite3 kütüphanesi kullanımı kolay ve birkaç aşama ile işlerinizi yapabileceğiniz bir kütüphanedir. **sqlite3** kütüphanesini yüklemek için komut satırına aşağıdakileri yazın.

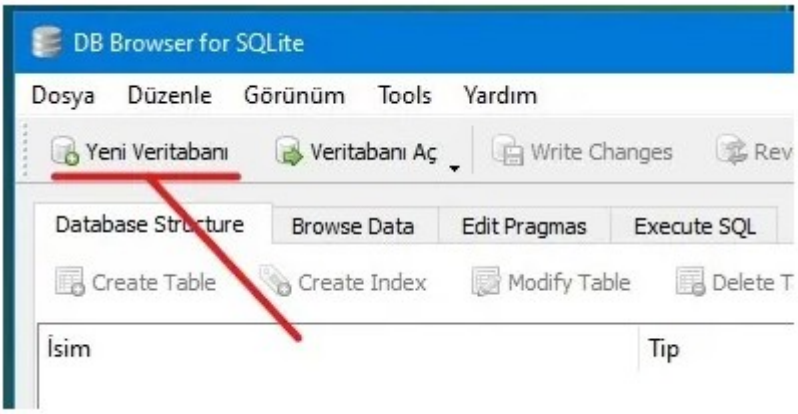
```
go get github.com/mattn/go-sqlite3
```

Tablo oluşturma ve düzenleme işlemlerinde bize kolaylık sağlaması için **DB Browser** programına ihtiyacımız olacak. Böylece hızlı bir şekilde veritabanı olaylarına geçiş yapmış olacağız.

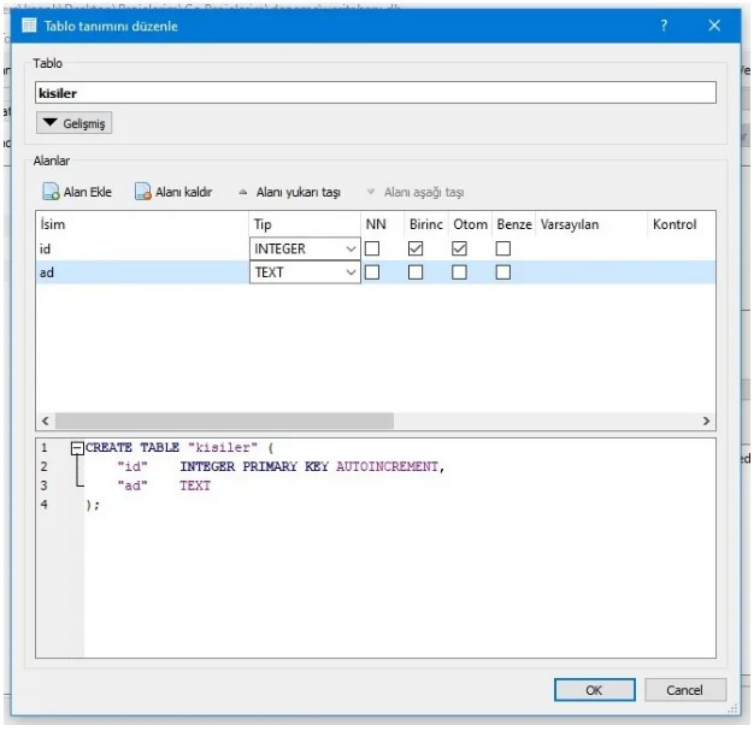
DB Browser programını aşağıdaki adresten indirebilirsiniz.
https://sqlitebrowser.org/dl/

Linux sistemlerin çoğunda uygulama deposunda bulunan bir uygulamadır.

Programımızı açıp sol üst taraftan **Yeni Veritabanı**'na tıklayalım.

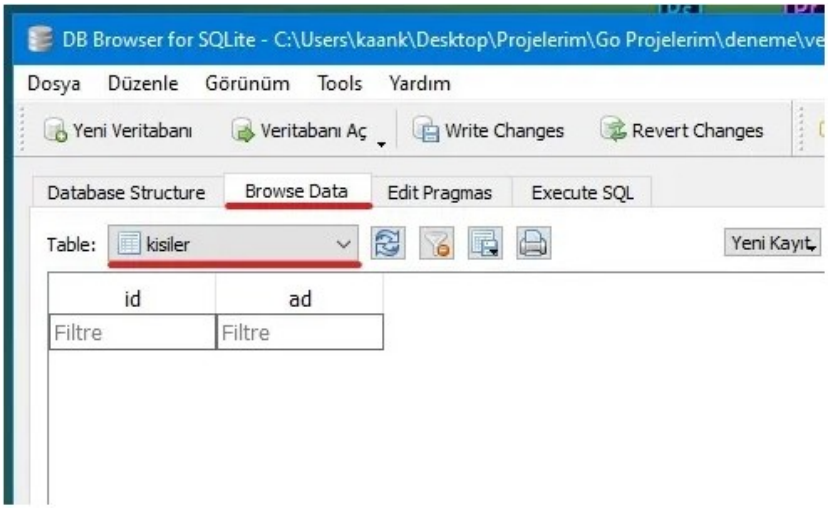


Veritabanının kayıt yerini, programımızın kodlarının bulunacağı **main.go** dosyası ile aynı yeri seçelim ve ismini **veritabanı.db** olarak kaydedelim. İstediğiniz ismi de verebilirsiniz.



Tablomuzun ismini **kisiler** olarak ayarlayalım. **Alan Ekle**'ye tıklayarak yukarıdaki gibi **id** ve **ad** isminde alanlar oluşturalım. **id** alanının tipini **INTEGER** yaparak, sayısal verileri saklayabilmesini sağlıyoruz. **Birincil Anahtar** ve **Otomatik Arttırma** bölümlerini seçiyoruz. Otomatik Arttırma özelliği sayesinde tabloya veri eklendiğinde **id** içindeki değer her eklemede artacaktır. Bu da her satır için ayırıcı bir özellik olacaktır. **ad** alanının tipini **TEXT** yapıyoruz. **OK** butonuna basarak tabloyu oluşturuyoruz.

Böylelikle içerisinde adları depolayabileceğimiz bir veritabanı olşturmuş olacağız. Oluşturduğumuz tablo her **ad** alanını belirterek veri ekleyişimizde o verinin yanındaki **id** alanına satıra özel numara verecektir.



Tablomuz içindeki kayıtları görmek için Browse Data sekmesine tıklayalım. Table kısmının yanında tablo oluştururken yazdığımız kisiler seçeneğini seçelim. Şuanlık tablomuz boş. Çünkü içine bir kayıtta bulunmadık. DB Browser programına bize yardımcı olduğu için teşekkür ederek artık Golang kodlama tarafına geçebiliriz.

sqlite3 Kütüphanesinin Kullanımı

main.go dosyamızı oluşturalım. Kütüphanelerimizi import edelim.

```
import (
   "database/sql"
   "fmt"
   _ "github.com/mattn/go-sqlite3"
)
```

database/sql ile Go üzerinde veritabanı işlemleri yapabiliyoruz. **fmt**'yi zaten biliyoruz. **github.com/mattn/go-sqlite3** ile de **sqlite3** kullanarak veritabanımızı yönetebiliriz. Buranın başına _ **(alt tire)** eklememizin sebebi vscode programı bu kütüphanenin kod içersinde kullanılmadığını düşünerek silmesini önlemek içindir. Basit şekilde veri tabanı bağlantısı nasıl yapılır görelim.

```
package main
import (
   "database/sql"
   "fmt"
   _ "github.com/mattn/go-sqlite3"
)
func main() {
   vt, _ := sql.Open("sqlite3", "./veritabanı.db") //veri tabanı dosyamız
   //Veri tabanı işlemleri için kodları yazacağımız bölüm
   vt.Close() //İşimiz bittikten sonra veri tabanımızı kapatıyoruz
}
```

sqlite3 Veri Ekleme İşlemi

```
func main() {
   vt, _ := sql.Open("sqlite3", "./veritabanı.db")
   işlem, _ := vt.Prepare("INSERT INTO kisiler(ad) values(?)")
   //Hangi bölüme eklenecekse yukarıda orayı belirtiyoruz
```

```
    veri, _ := işlem.Exec("Mustafa Kemal ATATÜRK") //Eklenecek değer
    id, _ := veri.LastInsertId() //Son girişin id numarısını aldık
    fmt.Println("Son kişinin id'si", id)
    vt.Close() //İşimiz bittikten sonra veri tabanımızı kapatıyoruz
}
```

sqlite3 Veri Güncelleme İşlemi

```
func main() {
    vt, _ := sql.Open("sqlite3", "./veritabanı.db")
    id:=1
    //değiştirilecek kısmın id numarası
    işlem, _ := vt.Prepare("update kisiler set ad=? where id=?")
    //Güncellenecek kısmı belirtiyoruz
    veri, _ := işlem.Exec("Gazi M. K. ATATÜRK", id)
    //Değişiklik ve Değiştirilen verinin id'si
    değişiklik, _ := veri.RowsAffected()
    fmt.Println("Değişen kişinin id'si: ", değişiklik)
    vt.Close() //İşimiz bittikten sonra veri tabanımızı kapatıyoruz
}
```

sqlite3 Veri Silme İşlemi

```
func main() {
    vt, _ := sql.Open("sqlite3", "./veritabanı.db")
    işlem, _ := vt.Prepare("delete from kisiler where id=?")
    //id numarasına göre sileceğiz
    veri, _ := işlem.Exec(id) //Silinecek kişinin id'si
    değişiklik, _ := veri.RowsAffected() //Silinen kişinin id'sini aldık
    fmt.Println("Silinen kişinin id'si: ", değişiklik)
    vt.Close()
}
```

sqlite3 Veri Sorgulama İşlemi

```
func main() {
    vt, _ := sql.Open("sqlite3", "./veritabanı.db")
    tablo, _ := vt.Query("SELECT * FROM kisiler")
    //Bu kısma sorguyu kullanıcağımız kodları yazacağız.
    tablo.Close() //İşimiz bittiği için tablo sorgulamayı kapattık
    vt.Close() //İşimiz bittikten sonra veri tabanımızı kapatıyoruz
}
```

Yukarıda **kisiler** tablosundaki tüm verileri sorgulamış olduk. Bir sonraki bölümde tablomuzdaki verileri nasıl kullanıcıya gösterebileceğimizi öğreneceğiz.

sqlite3 Verileri Sıralama/Gösterme İşlemi

```
func main() {
    vt, _ := sql.Open("sqlite3", "./veritabanı.db")
    tablo, _ := vt.Query("SELECT * FROM kisiler")
    var id int
    var ad string
    for tablo.Next() {
        aktarma := tablo.Scan(&id, &ad)
        //Tablo bölümlerini değişkenlere aktardık
        if aktarma == nil { //Boş mu kontol ediyoruz
            fmt.Println("Kişiler listesi boş")
```

```
        }else{
            //Boş değilse verileri ekrana bastırıyoruz
            fmt.Println(id, ad)
        }
    }
    tablo.Close() //İşimiz bittiği için tablo sorgulamayı kapattık
    vt.Close() //İşimiz bittikten sonra veri tabanımızı kapatıyoruz
}
```

MySQL Kütüphanesi

MYSQL, bir ilişkisel veritabanı yönetim sistemidir. MySQL yönetimi için kullanacağımız kütüphanenin adı **Go-MySQL-Driver**. Kütüphanemizi aşağıdaki gibi komut satırına yazarak indirelim.

```
go get -u github.com/go-sql-driver/mysql
```

MySQL paketlerimizi import edelim.

```
import "database/sql"
import _ "go-sql-driver/mysql"
```

MySQL Bağlantısını Yapma

Daha sonra **main()** fonksiyonumuz içerisinde MySQL bağlantımızı yapalım.

```
package main
import "database/sql"
import _ "go-sql-driver/mysql"
func main(){
        db, err := sql.Open("mysql", "kullanici:sifre@(127.0.0.1:3306)/vtismi?parseTime=true")
        err := db.Ping()
}
```

db adındaki fonksiyonel değişkenimize MySQL veritabanı bağlantı bilgilerimizi girdik. kullanici yeri MySQL kullanıcı adınızı, sifre yerine MySQL şifrenizi, 127.0.0.1:3306 yerine MySQL sunucunuzu e **vtismi** yerine de Veritabanı isminizi yazmayı unutmayın.

Daha sonra veritabanı bağlantı bilgilerimizi doğrulanmak için **db.Ping()** fonksiyonu ile bağlantı denemesi yolluyoruz. Bir hata ile karşılaşıldığında **err** değişkeninin içine hata çıktısını kaydedecektir.

Kolaylık olsun diye **main()** fonksiyonu dışına hata çıktılarını kontrol eden bir fonksiyon yazalım.

```
func kontrol(hata error){
    if hata != nil{
        log.Fatal(hata)
    }
}
```

Eğer hata çıktısı almak istemiyorsanız. **err** değişkeni yerine _ **(alt tire)** koyabilirsiniz. Aynen şu şekilde:

```
db, _ := sql.Open("mysql", "kullanici:sifre@(127.0.0.1:3306)/vtismi?parseTime=true")
```

İlk Tabloyu Oluşturma

Tablomuz şu şekilde olacak;

id	kullanici	sifre	tarih
1	kaan	1234	2019-08-10 12:30:00

Böyle bir tablo yapısını oluşturmak için aşağıdaki sorguyu çalıştırmamız gerekir.

```
CREATE TABLE kullanicilar (
    id INT AUTO_INCREMENT,
    kullanici TEXT NOT NULL,
    sifre TEXT NOT NULL,
    tarih DATETIME,
    PRIMARY KEY (id)
);
```

Bu sorguyu Golang tarafında yapmak istersek aşağıdaki gibi yazabiliriz.

```
sorgu := `
    CREATE TABLE kullanicilar (
        id INT AUTO_INCREMENT,
        kullanici TEXT NOT NULL,
        şifre TEXT NOT NULL,
        tarih DATETIME,
        PRIMARY KEY (id)
    );`
//Sorguyu çalıştırma
_, err := db.Exec(sorgu)
```

Bu işlemle birlikte MySQL veritabanımızda **kullanicilar** adında bir tablomuz oluşacaktır.

Tabloya Veri Girme

```
kullaniciDegeri := "johndoe"
sifreDegeri := "secret"
tarihDegeri := time.Now()
sonuc, err := db.Exec(`INSERT INTO kullanicilar (kullanici, sifre, tarih) VALUES (?, ?, ?)`, kullaniciDegeri, sifreDegeri, tarihDegeri)
kullaniciID, err := sonuc.LastInsertId()
fmt.Println("Eklenen kullanıcının id'si:", kullaniciID)
```

Tabloya Sorgu Yapma

```
//Tabloyu sorgulayıp sonuçları değişkenlere yazdıralım
sorgu:= `SELECT id, kullanici, sifre, tarih FROM kullanicilar WHERE id = ?`
err := db.QueryRow(sorgu, sorguid).Scan(&id, &kullanici, &sifre, &tarih)
//Çıkan aldığımız verileri ekrana bastıralım
fmt.Println(id, kulanici, sifre, tarih)
```

Tablodaki Tüm Verileri Sıralama

```
var (
    id int
    kullanici string
    sifre string
    tarih time.Time
)
tablo, _ := db.Query(`SELECT id, kullanici, sifre, tarih FROM kullanicilar`)
defer tablo.Close() //tabloyu kapamayı unutmuyoruz
for tablo.Next() {
    err := tablo.Scan(&id, &kullanici, &sifre, &tarih)
    kontrol(err)
    fmt.Println(id, kullanici, sifre, tarih) //kullaniciyi ekrana bastır
}
err := tablo.Err()
kontrol(err)
```

Eğer tablodaki verileri ekrana bastırmak yerine bir diziye (array) kaydetmek istiyorsak aşağıdaki gibi yapabiliriz.

```
type kullanici struct {
    id       int
    kullanici  string
    sifre  string
    tarih time.Time
}
```

```
tablo, _ := db.Query(`SELECT id, kullanici, sifre, tarih FROM kullanicilar`)
defer rows.Close()
var kullanicilar []kullanici
for tablo.Next() {
    var k kullanici
    err := tablo.Scan(&k.id, &k.kullanici, &k.sifre, &k.tarih)
    kontrol(err)
    kullanicilar = append(kullanicilar, k)
}
err := tablo.Err()
kontrol(err)
```

Bu işlemin sonucunda **kullanicilar** dizimiz şu şekilde olacaktır.

```
kullanicilar {
    kullanici {
        id:       1,
        kullanici:  "ahmet",
        sifre:  "1234",
        tarih: time.Time{wall: 0x0, ext: 63701044325, loc: (*time.Location)(nil)},
    },
    kullanici {
        id:       2,
        kullanici:  "mehmet",
        sifre:  "5678",
        tarih: time.Time{wall: 0x0, ext: 63701044622, loc: (*time.Location)(nil)},
    },
}
```

Tablodan Satır Silme

```
silineceksatir := 1
_, err := vt.Exec(`DELETE FROM kullanicilar WHERE id = ?`, silineceksatir)
kontrol(err)
```

Gördüğünüz gibi basit bir şekilde MySQL paketi ile veritabanı yönetimi yapabiliyoruz.

Hepsi Bir Arada

```
package main
import (
        "database/sql"
        "fmt"
        "log"
        "time"
        _ "github.com/go-sql-driver/mysql"
)
func kontrol(err error) {
        if err != nil {
                log.Fatal(err)
        }
}
func main() {
        vt, err := sql.Open("mysql", "kullanici:sifre@(127.0.0.1:3306)/vtismi?parseTime=true")
        if err := db.Ping(); err != nil {
                kontrol(err)
        }
```

```
	{ // Yeni tablo oluştur
		sorgu := `
	CREATE TABLE kullanicilar (
	   id INT AUTO_INCREMENT,
	   kullanici TEXT NOT NULL,
	   sifre TEXT NOT NULL,
	   tarih DATETIME,
	   PRIMARY KEY (id)
	);`
		_, err := db.Exec(sorgu)
		kontrol(err)
	}
	{ // Yeni kayıt ekle
		kullanici := "kaan"
		sifre := "1234"
		tarih := time.Now()
		sonuc, err := db.Exec(`INSERT INTO kullanicilar (kullanici, sifre, tarih) VALUES (?, ?, ?)`, username,
password, createdAt)
		kontrol(err)
		eklenenid, err := sonuc.LastInsertId()
		fmt.Println(eklenenid)
	}
	{ //  İstenilen kaydı sorgulama
		var (
			id       int
			kullanici string
			sifre    string
			tarih    time.Time
			sorguid  int = 1
		)
		sorgu := "SELECT id, kullanici, sifre, tarih FROM kullanici WHERE id = ?"
		err := vt.QueryRow(sorgu, sorguid).Scan(&id, &kullanici, &sifre, &tarih)
		kontrol(err)
		fmt.Println(id, kullanici, sifre, tarih)
	}
	{ // Tün kayıtları sorgula
		type kullanici struct {
			id       int
			kullanici string
			sifre    string
			tarih    time.Time
		}
		tablo, err := vt.Query(`SELECT id, kullanici, sifre, tarih FROM kullanicilar`)
		kontrol(err)
		defer tablo.Close()
		var kullanicilar []kullanici
		for tablo.Next() {
			var k kullanici
			err := tablo.Scan(&k.id, &k.kullanici, &k.sifre, &k.tarih)
			kontrol(err)
			kullanici = append(kullanicilar, k)
		}
		err := rows.Err()
		kontrol(err)
		fmt.Printf("%#v", kullanicilar)
	}
```

```
	//Kayıt Sil
	{
		silinecekid := 1
		_, err := vt.Exec(`DELETE FROM kullanicilar WHERE id = ?`, silinecekid)
		kontrol(err)
	}
}
```

WebView Kütüphanesi

WebView kütüphanesine giriş yapmadan önce bahsetmek istediğim birkaç konu var. Daha önce aramızda **Electron.JS**'i duyanlar olmuştur. Hani şu **Visual Studio Code**, **Skype**, **Atom**, **Discord** ve **Slack** gibi başarılı uygulamaların yazılmış olduğu Javascript kütüphanesinden bahsediyorum. **Electron.JS** ile yazılan uygulamalar **HTML**, **CSS** ve **Javascript**'in gücüyle kaliteli bir grafiksel kullanıcı arayüzüne ulaşabiliyor. Eğer bir **Web Developer**'sanız kolayca masaüstü uygulaması yazabiliyorsunuz. Ama **Electron.JS** ile yazılmış uygulamaların kötü yanları da var tabi. Uygulama boyutu bunlardan en sıkıntılı olanı. En basit bir uygulamanın boyutu **150 Megabyte** olabiliyor. Bir de **electron-packager** yardımı ile uygulama **build** edilirken uzun süre bekliyorsunuz.
Şimdi gelelim bizi bu olaylardan kurtaracak olan gözümün nuru Golang Kütüphanesi olan **WebView** kütüphanesine ♥

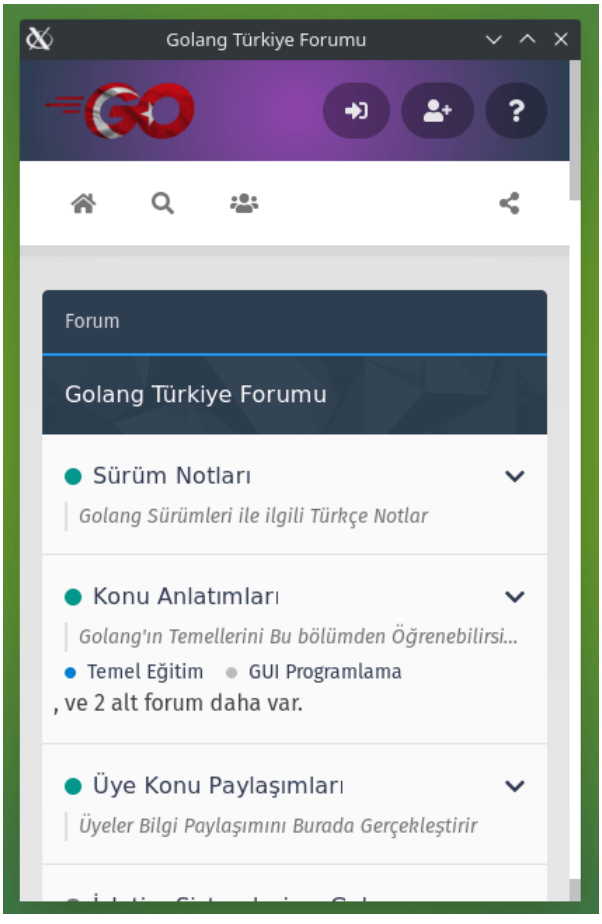
WebView kütüphanesi **zserge** arkadaşımız tarafından yazılmış olan, web sayfaları tasarlayıp programa dönüştürebildiğimiz, backend kısmını Golang rahatlığında yazdığımız bir kütüphane (veya paket)dir. **Build** işlemi sonrası aslında elimizde bir internet tarayıcısı olmuş oluyor. Bu tarayıcı üzerinden hazırlamış olduğumuz web sayfası görüntüleniyor. **Frontend** ve **Backend** arasındaki iletişimi ise **ExternalInvokeCallback** ile sağlıyoruz. Bu özelliği birazdan kodlar içerisinde açıklayacağım.

Sadece **Windows**, **GNU/Linux** ve **MacOS** için uygulama geliştirebiliyoruz.
GNU/Linux üzerinde **gtk-webkit2 (Bu paketin yüklü olması gerekir.)**, MacOS üzerinde **Cocoa/Webkit** ve Windows üzerinde **MSHTML** (IE10/11) alt yapısını kullanıyor. Bu detaylara bakacak olursak, Windows üzerinde çalışırken **Internet Explorer**'ı kullanacak. macOS ve GNU/Linux üzerinde ise Chrome benzeri bir altyapı kullanacak. Bu durumda GNU/Linux ve macOS için geliştirmek daha mantıklı çünkü daha fazla görsel efekt imkanı var olacaktır. Örnek: CSS3'teki -webkit- etiketi…
Gelelim kütüphanenin kurulumuna. Aşağıdaki komut ile kütüphanemizi indiriyoruz.

```
go get github.com/zserge/webview
```

Kütüphanemizi kurduğumuza göre ufak bir örnek görelim. Daha sonra detaylı örnekler göstereceğim.

```
package main
import "github.com/zserge/webview"
func main() {
    // Golang TR forumunu 400x600 boyuntunda boyutlandırılabilir olarak açar.
    webview.Open("Golang Türkiye Forumu",
        "https://golangtr.org", 400, 600, true)
}
```



Yukarıdaki gibi basit bir yöntem ile bir **gui** program oluşturabiliyorsunuz. Seviyeyi biraz yükseltelim ve sonraki örneğimize geçelim.

```
package main
import (
    "fmt"
    "net/http"
    "github.com/zserge/webview"
)
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Uygulamaya hoşgeldiniz!")
}
func serverOlustur() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":5555", nil)
}
func main() {
    go serverOlustur()
    webview.Open("Uygulama Başlığı",
        "http://localhost:5555", 400, 400, true)
}
```

Hemen açıklamasını yapayım. Kendi sunucumuzu oluşturmak için **"net/http"** kütüphanesini ekledik. **serverOlustur()** fonksiyonunda klasik web server oluşturmak için gerekli kodları yazdık. Görüntülenecek içeriği **handler()** fonksiyonunda belirttik.

main() fonksiyonu içerisindeki kodlarımıza geçelim. **serverOlustur()** fonksiyonunu **Goroutine** ile yazmazsak web server ayağa kaldırıldığında (açıldığında) kapanana kadar alt taraftaki Golang kodlarının çalışmasına sıra gelmez. Başına **go** ekleyerek aynı anda server'ın oluşturulmasına ve diğer kodların çalışmasını sağlıyoruz. **webview** kodlarımızda ise oluşturduğumuz web server'ın bilgilerini ve pencere ayarlarını giriyoruz.
Biraz değişiklikler ile istediğimiz bir klasörü göstermeye ayarlayabiliriz.

```
package main
import (
    "fmt"
    "net/http"
    "github.com/zserge/webview"
)
func serverOlustur() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":5555", nil)
}
func main() {
    klasor := http.FileServer(http.Dir("klasor/")) //Burada klasör yolunu belirtiyoruz
    http.Handle("/", http.StripPrefix("/", klasor))
    go serverOlustur()
    webview.Open("Uygulama Başlığı",
        "http://localhost:5555", 400, 400, true)
}
```

Daha ayrıntılı şekilde webview kodlarını şöyle de yazabiliriz.

```
    uygulama := webview.New(webview.Settings{
        Title:              "Golang Türkiye Forumu", //Pencere başlığı
        URL:                "https://golangtr.org", //Pencere adresi
        Width:              800, //Pencere genişliği
        Height:             600, //Pencere Yüksekliği
        Resizable:          false, //Boyutlandırma devredışı
    })
```

Aşağıdaki örneklerde daha açıklayıcı olması için bu şekilde yazacağız.
Sıra geldi Backend (Golang) ve Frontend (Javascript) arasındaki iletişimi sağlamaya.

Frontend'den Backend'e Veri Gönderme (Javascript ===> Golang)
Bu işlemi gerçekleştirebilmemiz için webview tarayıcısının frontend'deki sinyalleri dinlemesi gerekir. Bu özelliğin kullanımı şöyledir.

```
uygulama := webview.New(webview.Settings{
    Title:                "Golang Türkiye Forumu",
    URL:                  "https://golangtr.org",
    Width:                800,
    Height:               600,
    Resizable:            false,
    ExternalInvokeCallback: Yakala, //Buraya Dikkat - yanına () parantezler koymadan yazıyoruz.
})
```

ExternalInvokeCallback özelliğine **Yakala** yazdık. **Yakala** bizim tarayıcı sinyallerini dinleyeceğimiz fonksiyonumuz olacak. Sinyal deyince gözünüz korkmasın. Sinyal derken Frontend'den gönderilecek kelimeleri kastediyorum.
Yakala() fonksiyonumuz ise şöyle olacak.

```
func Yakala(uygulama webview.WebView, data string) {
    switch data {
        case "cikis":
            uygulama.Exit()
        case "tamekran":
            uygulama.SetFullscreen(true)
    }
}
```

Aslında kodlar kendi açıklıyor ama yine de açıklamasını yapayım.
Yakala() fonksiyonu parantezleri içerisine uygulama **webview.WebView** yazmamızın sebebi uygulama değişkeni üzerinden pencerede işlemler yapabilmektir. **data string** ile frontend'den aldığımız sinyalleri **data** değişkenine kaydedeceğiz.

switch yerine **if-else** kullanabilirdik fakat switch'in görünümü daha sade olduğundan switch'i tercih ettim. **data** değişkenine **"cikis"** yazılı sinyal geldiğinde **uygulama.Exit()** işlemini yapacak olduğunu görüyoruz. Golang tarafı bu kadar basit. Javascript tarafı daha da basit. Sinyalleri izlemeyi gördüğümüze göre nasıl sinyal gönderildiğini görelim. Örnek olarak butona tıkladığında sinyal gönderilmesini görelim.
HTML Kodumuz:

```
<button id="butonum">Çıkış</button>
```

Javascript Kodumuz:

```
function yazdir(yazi){
    document.getElementsByTagName("body").innerHTML(yazi)
}
```

Görüldüğü üzere bu kadar basit bir işlem.

Backend'den Frontend'e Veri Gönderme (Golang ===> Javascript)
Çok kolay bir işlemle bunu gerçekleştirebiliriz. Bu işe yarayan fonksiyonumuz **Eval()** fonksiyonu oluyor. Diyelim ki web sayfamızda aşağıdaki gibi bir Javascript kodu var.

```
function yazdir(yazi){
    document.getElementsByTagName("body").innerHTML(yazi)
}
```

Buradaki **yazdir()** fonksiyonunun görevi sayfadaki **body** etiketleri arasındaki içeriği değiştirmektir. Örnek olarak **yazdir("Merhaba")** denildiğinde sayfada **"Merhaba"** yazısı belirecektir.
Bizim bu Javascript fonksiyonunu Golang üzerinden tetiklememiz için Golang tarafına aşağıdaki kodu yazmamız gerekir.

```
uygulama.Eval("yazdir('Merhaba')")
```

İşte WebView kütüphanesi ile ilgili bir kaç ayar kodu:

```
Title string //Pencere başlığını ayarlama
URL string //Açılacak sayfanın URL'sini ayarlama
Width int //Açılacak pencerenin genişliğini ayarlama
Height int //Açılacak pencerenin yüksekliğini ayarlama
Resizable bool //Açılacak pencenin boyutu ayarlanabilir mi?
Debug bool //Normal tarayıcılarda F12'e basınca çıkan ekran aktif mi? (Sağ tıklayınca açılır)
ExternalInvokeCallback YakalaFonksiyonumuz //Sinyalleri yakalayan fonksiyonumuz girilir
```

İşte uygulama değişkenimize iliştirebileceğimiz fonksiyonlar:

```
Run() //Pencereyi başlatır
Loop(bloklama bool) //Ana pencerenin tek bir yinelemesini çalıştırır
SetTitle(başlık string) //Pencere başlığını değiştirir
SetFullscreen(true/false) //Penceyi tam ekran yapar veya küçültür
SetColor(r, g, b, a uint8) //Pencere arkaplan rengini ve saydamlığını ayarlar
Eval(js string) error //Pencereye Javascript kodu enjekte eder. İstenirse hata da döndürülebilir
InjectCSS(css string) //Pencereye CSS kodu enjekte eder
Dialog(diyalogtipi DialogType, flags int, başlık string, arg string) string
//Sistem diyalog penceresini gösterir ve string olarak çıktı verir
Terminate() //Ana pencere yinelemesini durdurur
Dispatch(func()) //Ana pencerede yürütülecek bazı işleri zamanlar
Exit() //Pencereyi kapatır ve kaynakları temizler
```

Gobot ile Arduino Yanıp-Sönen LED Yapımı

Bu yazımda sizlere Golang için Robotik kütüphanesi olan **Gobot**'tan bir örnek göstereceğim. Bu örneğimizde **Arduino**'da yanıp sönen LED yapacağız.
İlk olarak **Gobot** kütüphanesini indiriyoruz.

```
go get -d -u gobot.io/x/gobot/...
```

Daha sonra Arduino'muzla iletişim kurabilmemiz için **Gort**'u yüklememiz gerekiyor.
https://gort.io/documentation/getting_started/downloads/

Bu örnekte **Arduino Uno** kullanacağız. Arduino'muzun bilgisayarımıza bağlıyoruz ve hangi porta bağlı olduğunu öğrenmek için komut satırına aşağıdakileri yazıyoruz.

```
gort scan serial
```

Windows'ta **<COM*>** benzeri, Linux'ta ise **/dev/ttyUSB*** benzeri bir çıktı verecektir. Bu bizim Arduino'muzun bağlı olduğu portu gösteriyor.
Aşağıdaki kodlar yanıp sönen LED için yazılmıştır. Kodları gördükten sonra açıklamasını yapacağım.

```
package main
import (
	"time"
	"gobot.io/x/gobot"
	"gobot.io/x/gobot/drivers/gpio"
	"gobot.io/x/gobot/platforms/firmata"
)
func main() {
	firmataAdaptor := firmata.NewAdaptor("/dev/ttyUSB0")
	led := gpio.NewLedDriver(firmataAdaptor, "13")
	work := func() {
		gobot.Every(2*time.Second, func() {
			led.Toggle()
		})
	}
	robot := gobot.NewRobot("bot",
		[]gobot.Connection{firmataAdaptor},
		[]gobot.Device{led},
		work,
	)
	robot.Start()
}
```

Açıklamasına gelirsek;
Gobot ile alakalı kütüphanelerimizi ekliyoruz. **firmataAdaptor** değişkenimizde Arduino'muzun portunu yazıyoruz. Ben Linux kullandığım için Linux'taki portunu yazdım. **led** değişkenimizde ledimizin **13**. dijital pinde yer aldığını belirttik. Yani LED'imizin artı ucunu **13**. pine eksi ucunu ise **GND** (Ground-Toprak-Nötr) girişine bağlayacağız.
Sıra geldi çalışma fonksiyonumuz olan **work**'e. **work** değişkenine anonim bir fonksiyon tanımladık. Bu fonksiyonda **led.Toggle()** fonksiyonu ile her 2 saniyede yanıp-sönmesini ayarladık. En sondaki **robot** değişkeninde ise **firmataAdaptor** değişkenimizdeki **Arduino** portuyla bağlantı kurmasını ve hemen altında **led** değişkenini cihaz olarak tanıttık. Son olarak **work** değişkenindeki olayları gerçekleştirip, **robot.Start()** fonksiyonu ile çalışmasını sağladık.

Yukarıda gördüğünüz üzere **Firmata** kelimesini kullandık. **Firmata** bizim **Arduino** cihazımız ile iletişimde bulunabilmemizi sağlayan bir yazılım. Yukarıdaki kodlarımızın çalışması için Arduino'muz içerisine **Firmata** yazılımını yüklememiz gerekir. Onu da yüklemesi aşağıdaki gibi çok kolay bir işlem.

```
gort arduino upload firmata /dev/ttyUSB0
```

/dev/ttyUSB0 yerine kendi Arduino portunuzu yazmayı unutmayın.
Uygulamamızı başlatmak için ise aşağıdakileri yazıyoruz.

```
go run main.go
```

main.go yerine ne isim verdiyseniz onu yazınız.

Tinygo ile Küçük Yerler için Golang

Tinygo, Golang kodları ile mikro-denetleyicilere program yazmamızı sağlayan bir derleyicidir.

Aynı zamanda yazdığımız kodları mikro-denetleyicinin beynine flash eder. Flash etme kelimesinden kastım, beyne çalışacak kodları yazdırmaktır.

Gobot ile Arduino Yanıp-Sönen LED Yapımı konusunda bahsettiğim Gobot paketinden farkı, Gobot Firmata yazılımını Arduino'ya gömdükten sonra Arduino'ya çalıştırılabilir komutlar yolluyor. Yani kodlarımızı Arduino içine gömmediğinden, sadece Arduino USB veya TCP ile bağlı olduğundan çalışıyor.

Fakat **Tinygo**, Golang kodlarımızı Arduino'nun içerisine gömüyor. Bu sebeble Arduino'nun kodlarımızı çalıştırması için sadece bir elektrik kaynağına bağlı olması yetiyor.

Gelelim Kuruluma

GNU/Linux

Ubuntu/Debian
Birinci Adım:

```
wget https://github.com/tinygo-org/tinygo/releases/download/v0.9.0/tinygo_0.9.0_amd64.deb
```

İkinci Adım:

```
sudo dpkg -i tinygo_0.9.0_amd64.deb
```

Üçüncü Adım:

```
export PATH=$PATH:/usr/local/tinygo/bin
```

RaspBerry Pi
Birinci Adım:

```
wget https://github.com/tinygo-org/tinygo/releases/download/v0.9.0/tinygo_0.9.0_armhf.deb
```

İkinci Adım:

```
sudo dpkg -i tinygo_0.9.0_armhf.deb
```

Üçüncü Adım:

```
export PATH=$PATH:/usr/local/tinygo/bin
```

Arch Linux
Aur deposundan **tinygo-bin** olarak yükleyebilirsiniz.

Fedora Linux

```
sudo dnf install tinygo
```

Windows
Öncelikle şuanda Windows üzerinde deneme aşamasında olduğunu söylemeliyim.

İlk olarak LLVM 8'i kurmalısınız.

Buradan indirme sayfasına gidebilirsiniz.

LLVM 8 kurulumu esnasında "LLVM'yi ortam değişkenlerine ekle" seçeneğini seçmeyi unutmayın.

Daha sonra Tinygo arşiv dosyasını indirelim.

Tinygo Arşiv Dosyası İndir

Aşağıdaki komut ile Tinygo'yu kuralım.

```
PowerShell Expand-Archive -Path "c:\Downloads\tinygo0.9.0.windows-amd64.zip" -DestinationPath "c:\tinygo"
```

Aşağıdaki komut ile Tinygo'yu ortam değişkenlerine ekleyelim.

```
set PATH=%PATH%;C:\tinygo\bin;
```

MacOS
İlk adım:

```
brew tap tinygo-org/tools
```

İkinci Adım:

```
brew install tinygo
```

Kurulum Sonrası

Kurulum işlemlerimiz tamamlandıktan sonra kontrol etme amaçlı komut satırına aşağıdaki komutları yazalım.

```
tinygo version
```

Kullandığınız işletim sistemine göre fark göstermekle birlikte aşapıdakine benzer bir çıktı alacaksınız.

```
tinygo version 0.9.0 linux/amd64 (using go version go1.12.9)
```

Bu işlemler sırasında elimde bulunan **Arduino Uno** kartı ile işlemler yapacağımız belirteyim. Diğer kartlar ile arasında çok bir işlem farkı bulunmamaktadır. Aynı veya benzer yollardan sizce bu işlemleri gerçekleştirebilirsiniz.

Öncelikle Arduino Uno kartımızın hangi USB portuna bağlı olduğunu bulalım.

Windows üzerinden **COM3** benzeri bir portta takılıdır. İnternet üzerinden detaylı araştırma yapabilirsiniz.

Unix-like sistemlerde (Linux, MacOS) ise genelde **/dev/ttyUSB** veya **/dev/ttyACM** portarından birinde takılı olabilir. Arduino'nun bağlı olduğu portu **ls /dev/ttyUSB*** komutu ile öğrenebilirsiniz.

Ben **Arduino Uno** kartımın **/dev/ttyUSB0** üzerinde olduğu için aşağıdaki işlemlerimi ona göre yapacağım. Kullandığım komutları kendi portunuza göre değiştirmeyi unutmayın.

Aşağıda Arduino UNO üzerindeki **Built-In LED**'i saniyede bir yanıp-söndürmeye yarayan Golang kodları yer alıyor.

```
package main

import (
	"machine"
	"time"
)

func main() {
	led := machine.LED
	led.Configure(machine.PinConfig{Mode: machine.PinOutput})
	for {
		led.Low()
		time.Sleep(time.Millisecond * 1000)

		led.High()
		time.Sleep(time.Millisecond * 1000)
	}
}
```

Dosyamızın ismini **main.go** yapalım. Yukarıdaki Golang kodlarımızı kaydettikten sonra komut satırını **main.go** dosyasının bir üst klasöründe açalım.

Go kodlarımızı **Arduino** üzerine yazdırmak için aşağıdaki komutları kullanalım.

```
tinygo flash -target=arduino -port=/dev/ttyUSB0 ./kodumuzunbulunduğuklasör
```

Gördüğünüz gibi **Tinygo** ile flash etme işlemi çok basit.

Kullanıcı Adı, Ev Dizini Bilgilerini Alma

Golang'ta buna özel bir yerleşik kütüphane vardır. Kodların yanında açıklamaları bulunuyor.

```
package main
import (
    "fmt"
    "os"
    "os/user"
)
func main() {
    user, err := user.Current()
    if err != nil {
        panic(err)
    }
    // Mevcut Kullanıcı
    fmt.Println("Merhaba " + user.Name + " (id: " + user.Uid + ")")
    fmt.Println("Kullanıcı Adı: " + user.Username)
    fmt.Println("Ev Dizini: " + user.HomeDir)
    // Sudo kullanırken kullanıcı adı alma
    fmt.Println("Gerçek Kullanıcı: " + os.Getenv("SUDO_USER"))
}
```

Palindrom (Tersten Okunuşu) Fonksiyon

Palindrom nedir?
Palindrom (en: palindrome), bir kelimenin veya cümlenin tersten okunduğunda da aynı şekilde okunmasına denir.

Örnek: ey edip adanada pide ye = Palindrom'dur.

Golang'ta palindrom kelime veya cümleleri kontrol eden ve buna göre **bool** çıktı veren fonksiyon yazalım.

```
package main
import "fmt"
func palindrom(girdi string) bool {
	for i := 0; i < len(girdi)/2; i++ {
		if girdi != girdi[len(girdi)-i-1] {
			return false
		}
	}
	return true
}
func main() {
	fmt.Println(palindrom("ey edip adanada pide ye"))
	fmt.Println(palindrom("naber dostum"))
}
```

Çıktımız şöyle olacaktır.

```
true
false
```

gofmt Nedir?

Gofmt, Golang kaynak kodumuzu otomatik olarak biçimlendiren bir araçtır.

Gofmt ile düzenlenmiş kodun;

- Yazması kolay olur: ufak düzenlemelere takılmayız.
- Okuması kolay olur: düzenlenen kodlar tek stilde olacağından herkesin yazdığı kodların stili aynıdır.
- Sürdürmesi kolay olur: kaynak kodda yapılan değişiklikler sadece programın işleyişini değiştirir, sizin görüşünüzü değiştirmez.
- Tartışılmaz: kod yazma stilinizi kimse tartışmaz, münkü herkes aynı stilde yazar.

Kodumuzu biçimlendirelim!
Devam etmeden önce bilmemiz gereken şey, Golang geliştirmek için kullandığımız neredeyse tüm IDE'ler **gofmt** aracı ile kodlarımızın görünüşünü biçimlendirir.

Eğer kodlarınızı IDE kullanmadan yazıyorsanız, **gofmt** aracının kullanımını bilmeniz sizin için çok faydalı olacaktır. Örnek olarak aşağıdaki gibi bir Golang kaynak dosyamız olsun.

```
package main
func main() {
fmt.Println("Merhaba")
merhaba()
}
func merhaba(){
fmt.Println("Merhaba")
}
```

Gördüğünüz üzere yukarıdaki kodlarımızda herhangi bir boşluklandırma kullanılmamış. Kodumuz böyle olunca kangi kod hangi bloğun içinde zor belli oluyor. Hele ki daha uzun bir kod yazdığımızı düşünürsek, bu kodumuzu okumak yazmak ve sürdürmek daha zor bir hal alacaktır. İşte bu yüzden boşluklandırma işlemi çok önemlidir. Boşluklandırma işlemini **gofmt** aracı ile yapabiliriz.

Örnek kullanımı:

```
gofmt -w main.go
```

Yukarıdaki Golang kaynak dosyamızı **gofmt** aracı ile formatlandırırsak aşağıdaki gibi bir görüntüsü olur.

```
package main
func main() {
	fmt.Println("Merhaba")
	merhaba()
}
func merhaba() {
   fmt.Println("Merhaba")

}
```

Bu sayede daha okunabilir ve üstünde çalışması daha kolay bir koda sahip oluruz.

Go Geliştiricileri için Makefile

Makefile Nedir?
Makefile, çoğu komutu çalıştırmak için kullanabileceğimiz otomasyon aracıdır. Makefile'ı genellikle **Github** veya **Gitlab**'de programların ana dizininde bazı işlemleri otomatikleştirme için kullanıldığını görebilirsiniz.

Basit Bir Örnek
Bir proje klasörü oluşturalım ve bu klasörün içine makefile adında dosya oluşturalım. Daha sonra makefile dosyamızı herhangi bir editör ile açalım ve içerisine aşağıdakileri yazalım.

```
merhaba:
    echo "merhaba"
```

Gördüğünüz gibi programlama dillerine ve komutlara benzer bir yazılımı var.
Kodumuzu **make** komutu ile deneyebiliriz. Proje klasörümüzün içerisinde komut satırına **make merhaba** yazarak kodumuzun çıktısını görelim:

```
echo "Merhaba"
Merhaba
```

Gördüğünüz gibi **make** komutunun yanına **merhaba** ekleyerek **makefile** dosyamızdaki **merhaba** bölümünün çalışmasını sağladık. Makefile'ın genel mantığına baktığımızda komut satırı üzerinden yaptığımız işlemleri kısaltıyor.Şuanlık pek te birşeyi kısalttığı söylenemez. Sadece örnek için bunları yapıyoruz.

Basit Go Uygulaması İnşa Etme

```
package main
import "fmt"
func main() {
        fmt.Println("Merhaba")
}
```

Yukarıda gördüğünüz gibi basit bir **Go** uygulamamız var. Şimdi bu Go dosyamız ile işlem yapabilmek için **makefile** dosyamıza komutlar girelim.

```
merhaba:
        echo "Merhaba"
build:
        go build main.go
run:
        go run main.go
```

Yukarıda gördüğünüz gibi **makefile** dosyamıza bloklar açarak bildiğiniz komut satırı komutlarını girdik. Yukarıdaki kodların durumuna göre **make build** ile Go dosyamızı build ederiz ve **make run** ile Go dosyamızı çalıştırırız. Gayet basit bir mantığı var.

Peki bu olay bizim ne işimize yarayacak?
Örneğin bir projeyi 3 tane platform için build etmemiz gerekecek. Her platform için ayrı Go Ortamı bilgisi girmemiz gerekir. Hele ki build işlemini sürekli yapıyorsanız bu işten bıkabilirsiniz. Fakat **makefile** dosyasıyla işinizi kolaylaştırabilirsiniz.
Örneğimizi görelim:

```
derle:
        echo "Windows, Linux ve MacOS için Derleme İşlemi"
        GOOS=windows GOARCH=amd64 go build -o bin/main-windows64.exe main.go
        GOOS=linux GOARCH=amd64 go build -o bin/main-linux64 main.go
        GOOS=darwin GOARCH=amd64 go build -o bin/main-macos64 main.go
run:
        go run main.go
hepsi: derle run
```

derle bloğumuzun içerisine 3 platforma derlemek için komutlarımızı girdik. **run** bloğuna ise **main.go** dosyamızı çalıştırmak için komutumuzu girdik. **hepsi** bloğunun yanına ise **derle** ve **run** yazdık. Yani komut satırına **make hepsi** yazarsak hem derleme hem de çalıştırma işlemini yapacak.

Bu yazımızda Golang için **makefile** kullanımına örnek verdik. İlla Go'da kullanılacak diye bir kaide yok. Diğer programlama dillerinde veya komutlarınızı otomatize etmek istediğiniz zaman kullanabilirsiniz.

Derleme Detayını Görme

Golang'de normalde derleme işlemini yapmak için **go build** komutunu kullanırız. Bu komut terminal ekranından bize sadece bir hata olduğunda bilgi verir. Hata yoksa çalıştırılabilir dosyayı zaten oluşturur.

Peki programımızın derlenme esnasında bilgilendirmeyi nasıl görebiliriz?
İşte aşağıdaki gibi:

```
go build -gcflags=-m main.go
```

Yani build'e ek parametre olarak **-gcflags=-m** yazıyoruz. Nasıl gözüktüğünü örnek olarak görelim.

```
package main
import (
    "fmt"
    "os"
)
func main() {
        fmt.Println("Merhaba")
    fmt.Println(topla(2,2))
    os.Exit(0)
}
func topla(x,y int) int{
    return x + y
}
```

Yukarıdaki kodumuzun derleme çıktısı şöyle olacaktır.

```
# command-line-arguments
./main.go:13:6: can inline topla
./main.go:9:13: inlining call to fmt.Println
./main.go:10:22: inlining call to topla
./main.go:10:16: inlining call to fmt.Println
./main.go:9:14: "Merhaba" escapes to heap
./main.go:9:13: io.Writer(os.Stdout) escapes to heap
./main.go:10:16: io.Writer(os.Stdout) escapes to heap
./main.go:10:22: topla(2, 2) escapes to heap
./main.go:9:13: main []interface {} literal does not escape
./main.go:10:16: main []interface {} literal does not escape
:1: os.(*File).close .this does not escape
```

Kullanılan CPU Sayısını Öğrenme

Bu yazımızda Mevcut yapılan işlemde kullanılan mantıksal CPU sayısını öğrenmeyi göreceğiz. İşlemimiz çok basit.

Örneğimizi görelim:

```
package main
import (
	"fmt"
	"runtime"
)
func main() {
	fmt.Println(runtime.NumCPU())
}
```

Elemanın Dizideki İndeksini Bulma

Bu yazımızda sizlere belirli dizinde, değerini belirttiğimiz elemanın indeks numarasını bulacak bir fonksiyon yazacağız.

Örneğimizi görelim:

```
package main
import (
    "fmt"
)
func main() {
    dizi := []string{"Kaan", "Erkay", "Latif", "Ömer"}
    fmt.Println(dizi)
    //! Ekrana yazdırma işlemleri
    fmt.Println("Dizi uzunluğu:", len(dizi))
    fmt.Println("Erkay'ın dizideki indexi:", indeksBul(dizi, "Erkay"))
    fmt.Println("Latif'in dizideki indexi:", indeksBul(dizi, "Latif"))
    fmt.Println("Mehmet'in dizideki indexi:", indeksBul(dizi, "Mehmet"))
}
//! Dizideki indeksi bulacak olan fonksiyonumuz
func indeksBul(d []string, s string) int {
    var sıra int = -1
    for indeks, isim := range d {
        if isim == s {
            sıra = indeks
        }
    }
    return sıra
}
```

Gelelim açıklamasına:

dizi isminde içerisinde isimleri tutan **string** bir dizi oluşturduk.
Hemen aşağısında dizi ile alakalı bilgileri bastırmak için yazılmış **Println()** fonksiyonları bulunuyor.

indeksBul fonksiyonumuzun çalışma mantığını anlayalım. **indeksBul** fonksiyonu **d** adında **[]string** (string dizi) tipinde ve **s** adında **string** tipinde argümanlar alıyor.

[]string tarafına dizimizin **ismi**, **string** tarafına ise aradığımız **isim** gelecektir. Daha sonra **int** tipinde indeks numarasını çıktı olarak verecektir.

Fonksiyonumuzun içerisini incelediğimizde, **sıra** adında **int** tipinde **-1** değeri olan bir değişken ürettik. Bu değişkenin değerinin **-1** olmasının sebebi ilgili **indeks** bulunamadığında sonucu **-1** olarak döndürmesi için.

Daha sonra **for-range** ile **d** değişkenine gelen dizimizi taradık. **for** değişkeninin içine baktığımızda dizi taramasında **s** değişkeni (fonksiyona gelen string) ile **isim** (dizideki değerimiz) değişkeni birbirine eşitse sıra değişkenine indeks numarasını atamasını istedik.
Daha sonra **return** ile **sıra** değişkenini döndürdük. Bu sayede **Println()** fonksiyonunda **indeks** numarası görüntülenecektir.

KAYNAKÇA

Go Programlama Dili Hakkında Genel Bilgiler – Wikipedia

Go Programlama Dili Derleme Bilgileri – GoDoc

Go Programlama Dili Giriş Seviyesi – GoTour

Go Programlama Dili Operatörler – Go Bridge Uluslararası Topluluk Forumu

JSON Parsing – Latif Uluman

SON

Kitabımızın sonuna gelmiş bulunmaktayız. Kitap hakkında görüşlerinizi çok merak ediyorum. Umarım Go Programlama Dili'nin temellerini doğru düzgün anlatabilmişimdir.

Yazdığım kaynak hakkında görüşlerinizi, hataları ve önerilerinizi bildirmek için **GitHub** adresini aşağıya bıraktım. GitHub üzerinden kaynağın düzenlenmiş halini güncel tutmaya çalışacağım. GitHub deposuna Golang ile alakalı örnekler koymaya çalışacağım. Son versiyonu için **GitHub**'ı kontrol etmeyi unutmayın. Her zaman eleştiriye açığım. Bilginize..

Sosyal Medya hesaplarımdan benimle iletişim kurabilirsiniz.

Öneriler, fikirler ve aklınıza takılan soruları sorabilirsiniz. Çekingen olmanıza gerek yok, sohbet etmeyi severim :)

İLETİŞİM

 @ksckaan1  @ksckaan1  @ksckaan1  kaanksc@hotmail.com

Genellikle Twitter üzerinde aktifim. Kaynak hakkında anlık ilerlemelerime Twitter üzerinden şahit olabilirsiniz.

Github'dan yeni sürümü kontrol edebilirsiniz.

 https://github.com/ksckaan1/golangturkcekaynak