



Programlama Dili

Go Programlama Dili



[Bir Videoda Go Programlama Dilini Öğrenin Videosu için tıklayın.](#)

Platform **Id**

 Twitter [@ksckaan1](#)

 Telegram [@ksckaan1](#)

 Github [@ksckaan1](#)

ÖNEMLİ NOT

Güncel E-Kitap [buradaki adresten](#) yeni versiyonu indirmeyi unutmayın.

Katkıda bulunmak için yukarıdaki sosyal medya hesaplarımdan benimle iletişim kurabilirsiniz.

Keyifli Okumalar!

GİRİŞ

ÖNSÖZ

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.

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...

Katkıda Bulunanlar

Listede bulunan şahıslara az çok demeden katkıda bulundukları için çok teşekkür ederim.

**Liste harf sırasına göre yapılmıştır.*

[Adem Ali Durmuş](#) 

[Adem İlter](#) 

[Aykut Kardaş](#) 

[Berkay Çoban](#) 

[Erhan Yakut](#) 

[Latif Uluman](#) 

[Mert Şimşek](#) 

[Murat Mirgün Ercan](#) 

[Yusuf Turhan Papurcu](#) 

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.

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.

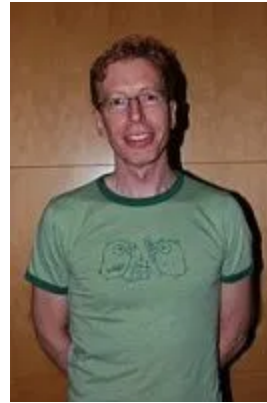
Bu adamların daha önceden bulunmuş olduğu projelere bakacak olursak Google'ın gerçekten bu kişileri cimbızla seçtiğini anlayabiliriz. İşte Golang'ın yaratıcılarının bulunduğu projeler:



Ken Thompson



Rob Pike



Robert Griesemer

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
- Tümüleşik 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!")
}
```

Go Derleyicisi Kurulumu

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

Buradan indirebilirsiniz

<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. Eğer son Go versiyonunu yüklemek istiyorsanız ve uygulama deponuzda son versiyon yoksa

<https://golang.org/dl/> adresinden Linux versiyonunu indirebilirsiniz.

Uygulama depolarından indirmek için:

GNU/Linux Dağıtımı Komut



`sudo apt install golang`

`sudo pacman -S go`

VSCode Go Eklentisi Yükleme

Golang'ı indirdiğimize göre bize Golang kodlarımızı yazacağımız bir Tümüleşik 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 indirebilirsiniz

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

```
git --version
```

Windows MacOS

Windows için: [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
```

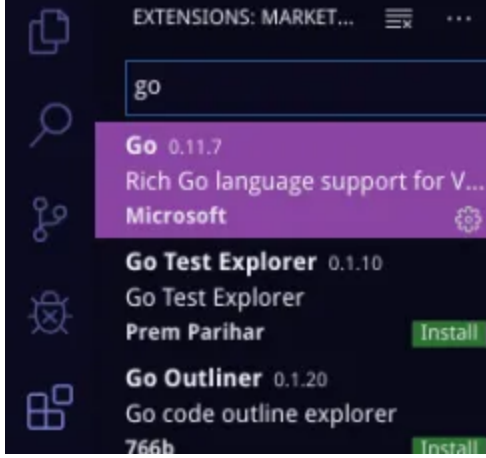
openSUSE

```
zypper install git
```

Mageia

```
urpmi git
```

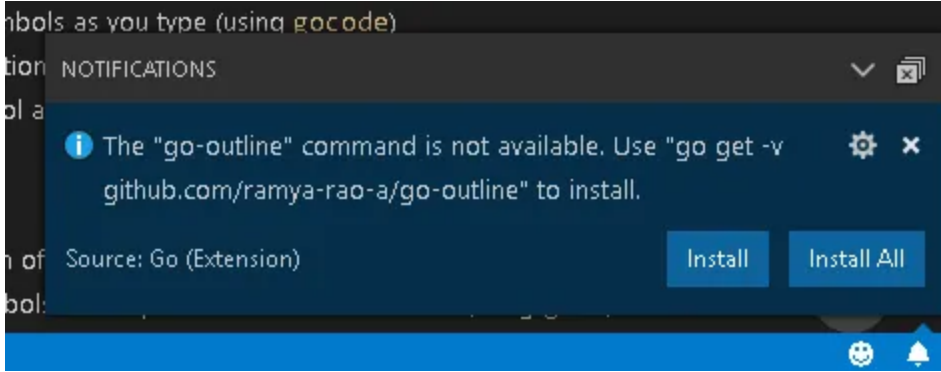
Git kurulumunu da yaptığımıza göre VSCode için Go eklentisini kurabiliriz.



VS Code Go Eklentisi

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.



VS Code Go Uyarısı

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 gelemiz.

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 geçmiş olmaya 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.

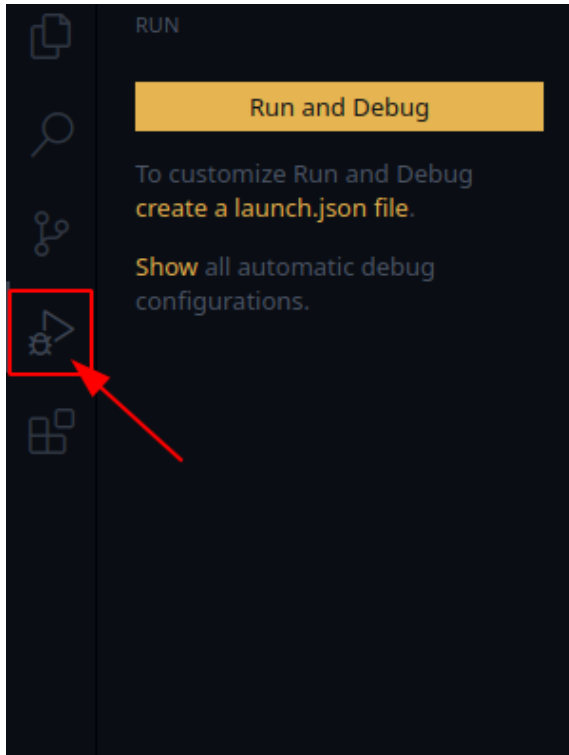
Linux üzerinden ise terminale derlenmiş main çıktınızın bulunduğu dizinde **./main** yazarak çalıştırabilirsiniz.

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.

VSCode Varsayılan Hata Ayıklayıcıyı Seçme

VSCode Go programlama yapıyorken, **F5** tuşuna bastığınızda üst tarafta hata ayıklayıcıyı seçmenizi ister. Her **F5** tuşuna bastığınızda hata ayıklayıcı seçim ekranı çıksın istemiyorsanız, yani her zaman bir hata ayıklayıcıyı kullansız istiyorsanız, yapacaklarınız çok basit.

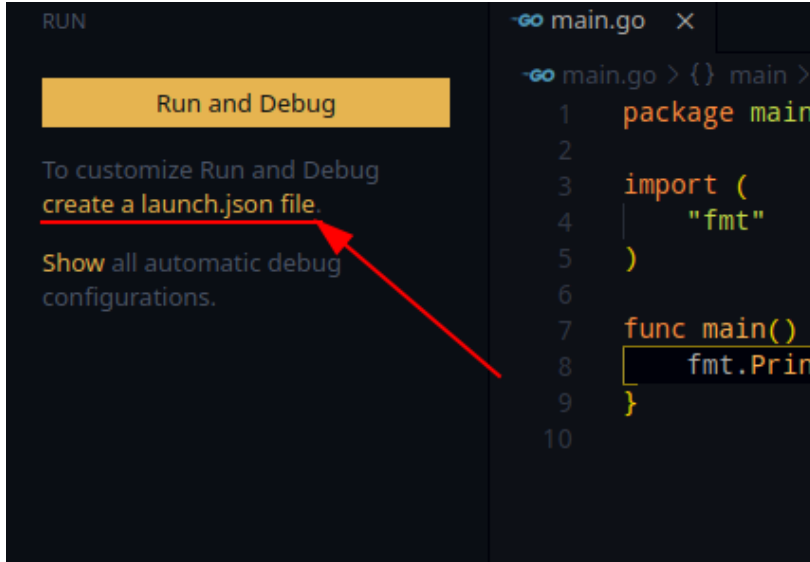
VSCode üzerinde ekranın sol tarafından Run sekmesine geçelim.



Adım.1

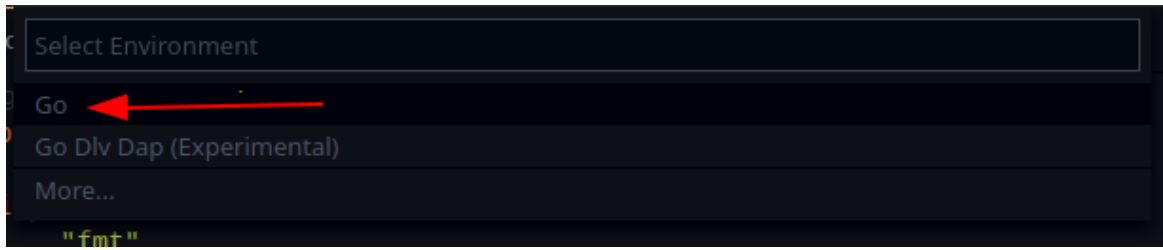
Biz varsayılan olarak **Go Hata Ayıklayıcısı**'nı seçeceğiz.

Go'yu varsayılan hale getirmek içinse, ekranın solundan create a launch.json file bağlantısına tıklayalım.



Adım.2

Üst tarafta açılan ekrandan **Go**'yu seçelim.



Adım.3

Seçtikten sonra VSCode bizim için `launch.json` adında bir dosya oluşturacak. Bu dosya **F5** tuşuna bastığımızda gerçekleşecek olayları barındırıyor. Dikkat edeceğimiz nokta type bölümünde `go` yazıyor olması.

```
{
  // Use IntelliSense to learn about possible attrib
  // Hover to view descriptions of existing attribut
  // For more information, visit: https://go.microso
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch",
      "type": "go",
      "request": "launch",
      "mode": "auto",
      "program": "${fileDirname}",
      "env": {},
      "args": []
    }
  ]
}
```

Adım.4

Daha sonra `launch.json` dosyamızı kaydedip kapatabiliriz.

Bir sonra hata ayıklama işleminde **Go** otomatik çalışacaktır.

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	arm64
netbsd	arm
openbsd	386
openbsd	arm64
openbsd	arm
plan9	386
plan9	amd64

GOOS	GOARCH
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 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.

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. Tümünüyle dosyaları paketlemek için ek kütüphaneler kullanabilirsiniz. Önerim **Statik** isimli kütüphanedir. İlerideki bölümlerde zaten bu kütüphaneyi kullanıyor olacağız.

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


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 işaretle 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 içerisinde 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 küsürlü sayıları tutar. Örnek: 3.14

Lütfen Dikkat!

Küsürlü 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 küsürlü (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 ingilizce”. 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

+ Toplar

- Çıkarır

* Çarpar

/ Böler

% Bölümden kalanı verir (Mod)

++ 1 arttırır

- 1 eksiltir

Örnek

2+5

10-3

3*4

10/2

10%3

1++

1-

İlişkisel Operatörler

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

Operatör Açıklama		Örnek
==	İki verinin eşitliği	2==2
!=	İki verinin eşitsizliği	2!=3
>	1. verinin 2. veriden büyüklüğü	5>3
<	1. verinin 2. veriden küçüklüğü	4<6
>=	1. verinin 2. veriden büyük veya eşitliği	4>=6
<=	1. verinin 2. veriden küçük veya eşitliği	3<=8

Mantıksal Operatörler

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

Operatör	Açıklama	Örnek
&&	VE operatörüdür. 2 koşulda doğruysa true değer verir	2==2 && 2==3 (false)
	VEYA operatörüdür. 2 koşuldan ikisi veya biri doğru ise true değer verir	2==2 2==3 (true)
!	DEĞİL operatörüdür. Koşul sonucunun tersini verir	!(2==2) (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
-=	Kendinden çıkarır	c-=2
=	Kendiyle çarpar	c=2
/=	Kendine böler	c/=2
%=	Kendine bölümünden kalanı atar	c%=3

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 bool = 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 küsürlü 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 anlaşılacağı 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. Şimdi de örneğimizin o versiyonunu görelim.

```
package main

import "fmt"
```

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

& (and) işareti ile deęişkenin bellekteki adresini öğrenebiliriz.

Çıktımız şöyle olacaktır;

iki

0xc00008c1d0

bir

0xc00008c1c0

Gördüğünüz gibi bellek adresi 2 sonuçta da farklı gözüküyor. Bu ikisinin de ayrı deęişkenler 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 değişkende 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 değişkenimizin 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, fonsiyonlarımızı oluşturmak için **func** anahtar kelimesini kullanırız. Yanına ise fonskiyonumuzun ismini yazarız. Parantez içine fonskiyonumuzun dışarıdan alacağı parametreler için değişken-tip tanımlaması yaparız. parantezin sağına ise fonskiyonun döndüreceği **return** değerinin tipini yazarız. Süslü parantezler içinde fonskiyonumuzun işlemleri bulunur. Son olarak return ile veri tipini belirlediğimiz değeri elde etmiş oluruz.

Main fonskiyonu içerisinde **topla(2,5)** fonskiyonu 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. Fonsiyonumuzun 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 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 içerisinde çekilmesini sağladık. Böylece fonksiyon bloğunun

sonundaki **return** kelimesinin yanına birşey yazmadık. Bu fonksiyonumuzun çıktısı ise **5 20** olacaktır.

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 toplama(sayilar ...int) int {
    toplam := 0
    for _, n := range sayilar {
        toplam += n
    }
    return toplam
}

func main() {
    fmt.Println(toplama(3, 4, 5, 6)) //18
}
```

Yukarıdaki fonksiyonumuzu inceleyelim. Vereceğimiz sayıları toplamaları 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 ögenin 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 bastırılmış 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**

kısımında 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 çarpımına o sayının faktöriyeli denir. Yani 4 sayısının faktöriyelini bulmak istiyorsak: $1_2_3 \times 4$ 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 sona gelene yani en küçük uint değeri olan 0'a dayanana kadar. Daha sonra sonucu main fonksiyonu içerisinde ekrana bastırdık.

Anonim Fonksiyonlar

Anonim fonksiyonların en büyük özelliği isimsiz olmalarıdır. (Zaten adından da belli oluyor ☺) Yazıldıkları yerde direkt olarak çalışırlar. Çalışırken diğer fonksiyonlardaki gibi parametre verilemediği için fonksiyonun sonuna parametre eklenerek çalıştırılırlar. Örneğimizi görelim:

```
package main

import "fmt"

func main() {
    metin := "Merhaba Dünya"

    func(a string) {
        fmt.Println(a)
    }(metin)
}
```


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.

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

```
package main

import "fmt"

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

Açıklaması:

For döngüsünden ayrı olarak **deger** adında **0** sayısal değerini alan bir değişken oluşturduk. **For** döngüsünde ise sadece koşul parametresini belirttik. 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

SADECE KOŞUL BELİRTEREK KULLANMA

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

```
package main

import "fmt"

func main() {
    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 belirttik. 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

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 akışında 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 deyimın 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

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

Sonraki Koşulu Kontrol Ettirme

Durumlar içerisinde kontrol etmemiz gereken başka durumlarda olabilir. Bunun için **fallthrough** deyimini kullanabiliriz.

```
package main

import "fmt"

func main() {
    x := 5
    switch {
        case x == 5:
            fmt.Println("x 5'tir")
            fallthrough
    }
```

```
        case x < 10:
            fmt.Println("x 10'dan küçüktür")
    }
}
```

Çıktımız aşağıdaki gibi olacaktır.

```
x 5'tir
x 10'dan küçüktür
```

Switch'e Özel Değişken Tanımlama

Tıpkı If deyimindeki Switch içerisinde kullanabileceğimiz değişkenler tanımlayabiliriz.

```
switch x := 5; {
    case x == 5:
        fmt.Println("x 5'tir")
    case x < 10:
        fmt.Println("x 10'dan küçüktür")
}
```

Defer

Defer kelimesinin Türkçe'deki karşılığı **ertelemektir**. Bu deyim 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** yazısını 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("İlk 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 alıştırmayı 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

Pointers (İşaretçiler)

İşaretçiler ile nesnenin bellekteki adresi üzerinden işlemler yapılabilir. Daha önceden işaretçileri içeren bir dil kullanmamış veya herhangi bir programlama dili de kullanmamış olabilirsiniz. Bu yüzden daha iyi anlamanız için işaretçilerin çalışma mantığını bilmemiz gerekir.

Örnek olarak `a` isminde bir değişkenimiz olsun. Bu değişkenimiz `tam sayı (int)` tipinde `8` değerini saklıyor olsun. Bu da Go dilinde aşağıdaki gibi oluyor.

```
var a int = 8
```

Bu değişkenimizi oluşturduktan sonra programımız çalışmaya başlayınca işletim sistemimiz bu değişkene özel, **bellek (RAM)** üzerinde bir alan ayıracaktır. Programın geri kalanında değişkenimizin değerine bu alan üzerinden ulaşılabilecektir. Yani bir değişken oluşturduğumuzda bellek üzerinde aşağıdaki resimdeki gibi bir alan oluştuğunu hayal edebilirsiniz.



Değişkenin RAM üzerindeki Alanı (Temsili)

Yukarıdaki resimde mor renkli olarak gördüğümüz ifade ise değişkenimizin bellekteki adresidir. (Bu adres temsildir. Zaten sürekli olarak değişen birşeydir.)

Peki Go'da işaretçileri nasıl kullanırız?

Bir program senaryosu belirleyelim. Bu programımızda yukarıdaki gibi `a` isminde `int` tipinde 8 değerini tutan bir değişkenimiz olsun. Ve `a` değişkenimize 5 ekleyen bir fonksiyonumuz olsun. Son olarak `a`'yı ekrana bastıralım.

```
package main

import "fmt"

func main() {
    a := 8
    ekle(a)
    fmt.Println(a)
    //sonucumuz yine 8 olacak
}

func ekle(v int) {
    v += 5
}
```

Yukarıdaki örnekte `a` değişkenini ekrana bastırdığımızda sonucun hala 8 olduğunu görüyoruz. Halbuki `ekle` fonksiyonunun içerisinde gördüğümün gibi 5 ekliyoruz.

`a` değişkeninin değişmeme sebebi şudur:
`ekle` fonksiyonunun parametresi olarak `int` tipinde `v` değişkenini oluşturduk. `v` değişkenimiz aslında `a`'dan gelen değeri kullanmamızı sağlıyor. Yani bize `a`'nın kendisini vermiyor. O yüzden `v` üzerinde değişiklik yaptığımızda `a`'ya yansımayacaktır.

a değişkenini değiştirebilmemiz için bize a'nın bellekteki adresi gerekiyor. Bunun için de & (ampersand) işaretini kullanabiliriz.

&a

0xc00017604

Ampersand kullanımı

Örnek vermek gerekirse:

```
func main() {  
    a := 8  
    fmt.Println(&a) //Çıktımız: 0xc0000b8010  
}
```

Artık a'nın bellekteki adresini öğrenebiliyoruz. Sıra geldi bu adres üzerinden a'nın değerine ulaşabilmeye.

Bunun için de * (yıldız) işaretini kullanabiliriz.

```
func main() {  
    a := 8  
    fmt.Println(&a) //Çıktımız: 0xc0000b8010  
    b := &a  
    fmt.Println(b) //Çıktımız: 0xc0000b8010  
    fmt.Println(*b) //Çıktımız: 8  
}
```

Yukarıdaki örneği incelediğimizde b değişkenine a'nın adresini atadık. b'yi bastırdığımızda a'nın bellekteki adresini görebiliriz. Aynı zamanda *b şeklinde kullanarak a'nın içindeki değere de ulaşabiliriz. Bu durumda a ve b değişkenleri aynı bellek alanını temsil ediyorlar.

a değişkenine b üzerinden değişiklik yapmak için aşağıdaki yöntemi uygulayabilirsiniz.

```
package main

import "fmt"

func main() {
    a := 8
    b := &a
    *b = 10
    fmt.Println(a) //10
}
```

En başta kurguladığımız senaryoyu işaretçiler ile kolayca yapabiliriz.

```
package main

import "fmt"

func main() {
    a := 8
    ekle(&a)
    fmt.Println(a) //Çıktımız: 13
}

func ekle(v *int) {
    *v += 5
}
```

Yukarıdaki örneği incelediğimizde, `ekle` fonksiyonumuzu oluştururken parametre olarak verdiğimiz `v` değişkeninin tipinde önce `*` işareti koyduk. Bunun sebebi `v` değişkeni ile fonksiyonumuza gelecek olan adresin içindeki değere ulaşabilmektir.

`ekle` fonksiyonunun içerisindeki `v`'yi kullanırken de başına `*` koyarak kullandık.

`main` fonksiyonumuzda `ekle` fonksiyonunu çağırırken de `a` değişkenini `&` işareti kullanarak bellekteki adresi ile verdik.

Bu sayede `a` değişkenine bellekteki adresi ile müdahale etmiş olduk.

Struct

Go programlama dilinde sınıflar yoktur. Sınıflar yerine struct'lar (yapılar) vardır. Yapılar sayesinde bir nesne oluşturabilir ve bu nesneye ait özellikler oluşturabiliriz. Örnek bir struct oluşturalım.

```
{% code title="struct örneği" %}
```

```
type kişi struct {  
    isim      string  
    soyİsim   string  
    yaş       int  
}
```

```
{% endcode %}
```

type terimi ile yeni bir tür oluşturabiliyoruz. İsmi `kişi` olarak verdik ve türünün de `struct` olacağını söyledik. Yukarıdaki şekilde bir yapı oluşturmuş olduk. Bu yapı içerisinde `isim`, `soyİsim` ve `yaş` değişkenlerine sahip. Yukarıdaki yapı üzerinden bir nesne örneği oluşturduğumuzda örneğimiz bu değişkenlere sahip olacak.

```
{% code title="Örnek Kullanım:" %}
```

```
package main
```

```
import "fmt"
```

```
type kişi struct {  
    isim      string  
    soyİsim   string  
    yaş       int  
}
```

```
func main() {
```

```
kişil := kişi{"Kaan", "Kuşcu", 23}  
  
fmt.Println(kişil)  
  
}
```

{% endcode %}

main() fonksiyonunun içerisinde incelediğimizde, `kişil` isminde `kişi{}` yapısında bir nesne örneği oluşturuyoruz. İçerisine oluşturucu parametreler olarak `kişi struct`'ındaki sıralamayı göz önünde bulundurarak parametrelerimi giriyoruz. Daha sonra `kişil` nesne örneğini ekrana bastırıyoruz. Çıktımız aşağıdaki gibi olacaktır:

```
{Kaan Kuşcu 23}
```

Yukarıdaki örnekte nesneyi tanımlama sırasında değer atamasını yaptık. Nesnenin alt değişkenlerine ulaşarak da tanımlama yapabiliriz.

```
kişil := kişi{"Kaan", "Kuşcu", 23}  
kişil.isim = "Ahmet"  
kişil.soyİsim = "Karaca"  
kişil.yaş = 34  
  
fmt.Println(kişil) //{Ahmet Karaca 34}
```

Nesne örneğini oluştururken parametreleri boş bırakıp sonradan da atama yapabiliriz.

```
kişil := kişi{  
kişil.isim, kişil.soyİsim = "M. K.", "ATATÜRK"  
kişil.yaş = 999  
  
fmt.Println(kişil) //{M. K. ATATÜRK 999}
```

İsim Belirterek Tanımlama

Nesneye özel değişkenleri tanımlarken değişken ismini belirterek de tanımlama yapabiliriz.

```
kişil := kişi{soyİsim: "Kuşcu", isim: "Kaan", yaş: 23}
```

```
fmt.Println(kişil) //{Kaan Kuşcu 23}
```

Değişken ismini belirterek atama yaptığımız için sıralamaya dikkat etmemiz gerekli değildir.

Anonim Struct'lar

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 Fonksiyonlar (Methodlar)

Bu bölümde bir struct'a özel nasıl fonksiyon oluşturacağımızı göreceğiz.

Örneğimizi görelim:

```
package main

import "fmt"

type insan struct {
    isim string
    yaş  int
}

func (i insan) tanıt() {
    fmt.Printf("Merhaba, Ben %s. %d yaşındayım.", i.isim, i.yaş)
}

func main() {
    kişi := insan{"Kaan", 23}
    kişi.tanıt()
}
```

`insan` isminde bir struct tipi oluşturduk. Bu yapımızın tıpkı insanlarda olduğu gibi `isim` ve `yaş` değişkenleri var.

Hemen aşağısında bir fonksiyon oluşturduk. Bu fonksiyonumuzun özelliği ise fonksiyonun isminden önce parantez içerisinde hangi struct'ta çalışacağını belirtmemizdir. `insan` struct'ının içerisindeki değişkenlere ise `i` değişkeni ile eriştik.

Daha sonra `main` fonksiyonumuzda `kişi` isminde `insan` tipinde bir nesne oluşturduk. `kişi.tanıt()` yazarak `insan` struct

tipinde oluşturduğumuz nesne için olan tanıt
fonksiyonumuzu çalıştırdık.

Çıktımızı görelim:

Merhaba, Ben Kaan. 23 yaşındayım.

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** indeksine **“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 olacaktır.

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 bastırduğumuzda 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 diziye **for** döngüsü ile tekrarlayabilir. Bir dilim range edildiğinde, tekrarlama başına iki değer döndürür (return). Birinci değer dizinin **indeksi**, ikinci değer ise bu indeksin içindeki **değerdir**. 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 dizimizdeki 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.

1. indeks = Ali
2. indeks = Veli
3. indeks = Hasan
4. indeks = Ahmet
5. 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"},
    "kadınlar": insan{"Ayşe", "Fatma", "Hayriye"},
}
func main() {
    fmt.Println(m["erkekler"])
    fmt.Println(m["kadınlar"])
    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 bölgeyi bir dizi olduğunu belirttik. Dizinin içerisine **"erkekler"** isiminde **insan** tipinde bir bölge oluşturduk ve içine **3** tane **string** tipinde değerimizi girdik. Aynı işlemi **"kadınlar"** isimli bölge içinde yaptık. **main** fonksiyonumuz içerisinde **erkekler** ve **kadınlar** 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} kadınlar:{Ayşe Fatma Hayriye}]
```

Burada ayrıntıyı farkedelim. **m** dizisini ekrana bastırdığımızda map yeni bölgeyi 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)

Interface'in Go dili üzerindeki kullanımını basitçe açıklayalım. Interface struct nesnelerin, struct tipine göre ilişkili fonksiyonların çalışmasını sağlar. Detayına inmek için önce interface'in nasıl oluşturulduğuna bakalım.

```
{% code title="interface oluşturma" %}
```

```
type hesap interface{  
    hesapla()  
}
```

```
{% endcode %}
```

Yukarıda ne yaptığımıza bakacak olursak, type ile hesap isminde bir interface oluşturduk. hesapla() ise hesap interface'imiz ile ilişkili olacak fonksiyonumuzun ismi olacak.

Interface'in belirli structlar üzerinde etki göstermesi gerekiyor. Bu struct'ları da oluşturalım.

```
type toplam struct {  
    sayı1 int  
    sayı2 int  
}
```

```
type çarpım struct{  
    sayı1 int  
    sayı2 int  
}
```

Yukarıdaki yapılarla toplanılacak ve çarpılacak sayıları barındıran nesneler oluşturacağız. Sonrasında bu yapılara iliştilen struct fonksiyonlar yazacağız.

Örnek olarak:

```
işlem1 := toplam{5,10}
```

```
işlem2 := çarpım{5,10}
```

Şimdi de bu structlar için fonksiyonlar oluşturalım.

```
func (t *toplam) hesapla() {  
    fmt.Println(t.sayı1 + t.sayı2)  
}
```

```
func (ç *çarpım) hesapla() {  
    fmt.Println(ç.sayı1 + ç.sayı2)  
}
```

Yukarıdaki oluşturduğumuz fonksiyonlarda dikkat edilmesi gereken nokta iki struct fonksiyonun da ismi interface içerisinde belirttiğimiz gibi hesapla olmasıdır.

İki fonksiyonda ismini aynı yapmamızın sebebi: oluşturduğumuz interface, nesnenin tipine göre hesapla fonksiyonunu çalıştırmasıdır. Yani işlem1 nesnesini hesap interface'i ile karşılaştırıldığında toplam struct'ı olduğunu algılayıp, toplam struct'ı ile ilişkili hesapla fonksiyonu çalışacaktır. Biraz karışık bir cümle olduğunun farkındayım. O yüzden işlem yaparak öğrenebiliriz.

İlk olarak interface'imizi parametre olarak alan bir fonksiyon oluşturalım.

```
func hesapYap(h hesap){  
    h.hesapla()  
}
```

Yukarıda yaptığımız işlem çok basit. hesap interface tipini h değişkeni ile çağırdık. h.hesapla() ile fonksiyonumuzu çalıştırdık.

Gelelim interfacemizi nasıl kullandığımıza:

```
package main

import "fmt"

type hesap interface {
    hesapla()
}

type toplam struct {
    sayı1 int
    sayı2 int
}

type çarpım struct {
    sayı1 int
    sayı2 int
}

func (t *toplam) hesapla() {
    fmt.Println(t.sayı1 + t.sayı2)
}

func (ç *çarpım) hesapla() {
    fmt.Println(ç.sayı1 * ç.sayı2)
}

func hesapYap(h hesap) {
    h.hesapla()
}

func main() {
    işlem1 := toplam{5, 10}

    işlem2 := çarpım{5, 10}

    //hesap interface'inden bir örnek oluşturalım
    var işlem hesap

    //işlem1'in adresini işlem interface'ine atayalım.
    işlem = &işlem1

    //interface toplam structı olduğunu algılayıp toplama
    işlemi yapacaktır.
    hesapYap((işlem))
}
```

```
//işlem2'nin adresini işlem interface'ine atayalım.  
işlem = &işlem2
```

```
//interface çarpım structı olduğunu algılayıp çarpma işlemi  
yapacaktır.  
    hesapYap((işlem))  
}
```

Özet geçmek gerekirse, en yukarıda interface'in tanımını yaptığım cümleyi aşağıya kopyala + yapıştır yapayım.

Interface struct nesnelerin, struct tipine göre ilişkili fonksiyonların çalışmasını sağlar.

Goroutine

Goroutine'ler **Go Runtime** tarafından yönetilen hafif bir sistemdir. Bir işlemi eşzamanlı olarak yapmak istiyorsak, Goroutine'den faydalanabiliriz. Bu sayede aynı çalışma-zamanı içerisinde birden fazla iş parçacığı oluşturabiliriz.

Terimler

Ana iş parçacığı

Main() fonksiyonu içerisine yazdığımız, asenkron olmayan kodlardır. Varsayılan olarak Go Runtime bu iş parçacığını izler. Programımız asenkron işlemlerin tamamlanmasını beklemiyorsa, ana iş parçacığı tamamlandığında program sona erer.

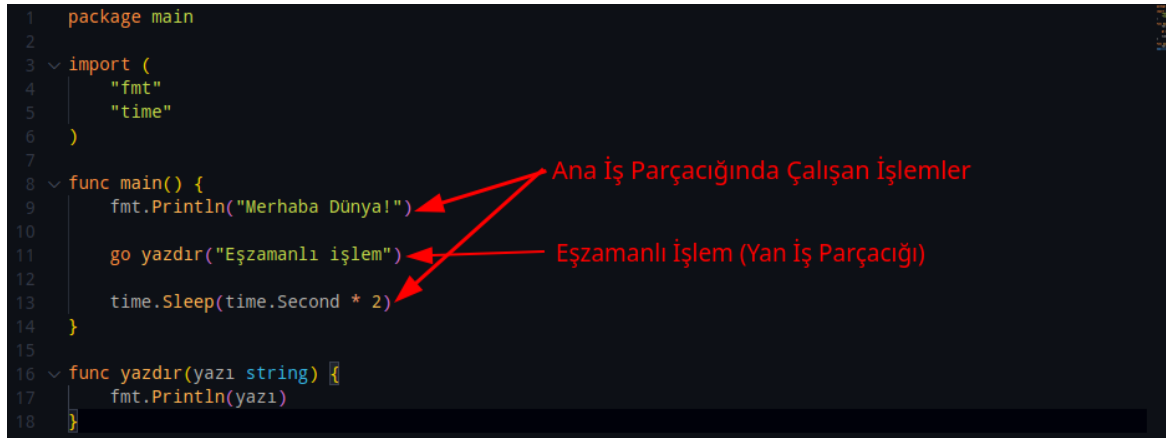
Eşzamanlılık

Eşzamanlılık, programlamada bir işlem gerçekleşirken, aynı zamanda başka işlemlerin de gerçekleşmesidir.

Eşzamanlı Bir İşlem Oluşturalım

Eşzamanlı bir işlem oluşturmak için go anahtar kelimesinden faydalanabiliriz. Bunun için eşzamanlı çalışacak işlemin başına go yazmamız yeterli olacaktır.

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     fmt.Println("Merhaba Dünya!")
10
11     go yazdır("Eşzamanlı işlem")
12
13     time.Sleep(time.Second * 2)
14 }
15
16 func yazdır(yazı string) {
17     fmt.Println(yazı)
18 }
```



Asenkron İşlem Örneği

Aslında yukarıdaki örnekte `time.Sleep()` kullanarak 2 saniye bekletmemizin bir sebebi. Eğer `time.Sleep()` eklememiş olsaydık, ekrana *"Merhaba Dünya!"* yazıldıktan sonra programımız sonlanacaktı. Bunun sebebi Go Runtime'ının Sadece Ana iş parçacığını beklemesi. Ana iş parçacığındaki işlemler sonlandıktan sonra, diğer işlemleri beklemiyor. Yukarıdaki örnekte bunu engellemek için `time.Sleep()` kullandık. Böylece program 2 saniye beklerken eşzamanlı işlemimiz de tamamlandı. Tabii `time.Sleep()` kullanarak beklemek mantıklı bir yöntem değil. İşlemin ne kadar süreceğini bilmediğimiz durumlar olacaktır. Bunun için Kanalları kullanabiliriz.

Kanallar (Channels)

Kanallar, Go dilinde asenkron programlama yaparken değer aktarımı yapabileceğimiz hatlardır. Kanala değer atanması iş parçacığı tarafından bekleneneği için asenkron işlemler arasındaki senkronizasyonu ayarlayabiliriz. Kanallar `make()` fonksiyonu ile oluşturulur.

```
{% code title="Örnek" %}
```

```
k := make(chan bool)
```

```
{% endcode %}
```

Yukarıdaki örnekte `make()` fonksiyonu ile `k` isminde bir kanal oluşturduk. Bu kanalın özelliği `bool` tipinde değer taşımasıdır. Yani bu kanal ile `true` veya `false` değerlerini taşıyabiliriz. Kanala değer göndermek için `<-` işaretini kullanırız. Yani bir nevi atama işlemi yapıyoruz. Atama işleminden farkı, kanala atama işlemi yapılana kadar iş parçacığının devam etmemesidir.

```
{% code title="Örnek Atama" %}
```

```
k <- true
```

```
{% endcode %}
```

Atama işlemi ile kanalımıza değer yolladık. Bir de bu kanalın çıkış noktası olması gerekir. Bu çıkış noktasında, ister kanaldan gelen veriyi bir değişkene atayabiliriz, istersek de sadece kanala veri gelmesini bekleyebiliriz.

```
{% code title="Kanaldan gelen değeri değişkene atama" %}
```

```
a := <-k
```



```
{% endcode %}
```

Yukarıdaki örnekte `a` isimli değişkene `k` kanalından gelen `bool` tipinde değer atadık. `a` değişkenine atama işlemi `k` kanalına değer gönderildiği zaman yapılacaktır. Yani `k` kanalına değer gelene kadar iş parçacığı duraklatılacaktır. (*Program `k` kanalına gelecek değeri bekler.*)

```
{% code title="Sadece kanala değer gelmesini beklemek" %}
```

```
<- k
```

```
{% endcode %}
```

Yukarıdaki anlatılanlardan yola çıkarak bir örnek oluşturalım.

```
{% code title="Örnek kanal işlemleri" %}
```

```
package main
```

```
import (  
    "time"  
)
```

```
func main() {
```

```
    //bir kanal oluşturalım  
    k := make(chan bool)  
    //bu kanalımız bool değer taşıyacak
```

```
    //asenكرون bir iş parçacığı oluşturalım  
    go func() {
```

```
        //bu iş parçacığı 5 sn beklesin  
        time.Sleep(time.Second * 5)
```

```
        //k kanalına bool bir değer gönderelim  
        k <- true
```

```
    }()
```

```
    //ana iş parçacığı k kanalına değer gelene kadar bekleyecek
```

```
    <-k
    //değer geldiğinde program sonlanacaktır.
}
```

```
{% endcode %}
```

Boyutlu Kanal Oluşturma

Oluşturduğumuz kanala boyut vermek de mümkün. Yani kanalımıza birden fazla değer yollayabiliyoruz. Bunun için kanalı oluştururken `make()` fonksiyonunda boyutu da belirtelim.

```
{% code title="Örnek" %}
```

```
package main

import (
    "fmt"
    "time"
)

func main() {

    //2 adet bool değer taşıyan bir kanal oluşturalım
    k := make(chan bool, 2)

    //asenkron bir iş parçacığı oluşturalım
    go func() {

        //5 sn beklesin
        time.Sleep(time.Second * 5)

        //k kanalına bool bir değer gönderelim
        k <- true

        //tekrardan 2 sn beklesin
        time.Sleep(time.Second * 2)

        //ve k kanalına 2. değer de gönderilsin.
        k <- false
    }()
}
```

```
    }()  
  
    //ana iş parçacığı k kanalına 2 değer gelene kadar  
    bekleyecek  
    fmt.Println(<-k, <-k) //çıktı: true false  
    //iki bool değeri de bastırmak için k kanalını 2 defa  
    yazdık  
    }  
  
{% endcode %}
```

Ana iş parçacığı (*main()* içerisinde yazılan kodlar) devam etmek için k kanalına gelen 2 değeri de bekleyecektir.

fmt.Println() içerisinde sadece bir defa <-k yazsaydık, k kanalına ilk gelen değeri ekrana bastıracaktı.

Anonim Goroutine Fonksiyonlar

Bu yazımız **Goroutine** ve **Kanallar** dersi için biraz alıştırmada olacak.

Daha önceki yazılarımızda belirli bir fonksiyonu **Goroutine** ile **asenكرون** (eş zamanlı) olarak çalıştırmayı gördük. Bu yazımızda da **anonim** bir Goroutine fonksiyonunu göreceğiz. Bu fonksiyonun özelliği bir ismi olmaması ve asenكرون 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. Çünkü fonksiyonumuza parametreleri bu parantezler içerisinde yolluyoruz. Şuanlık parametre yollamadığımızın için boş kalacak. Bu fonksiyonumuz programın geri kalanı ile aynı zamanda çalışacak. Hatta programın geri kalanı ile bağlantısı bile olmayacak. Bu sebepten ötürü mantiken 2 saniye sonra işlem yapmasını belirttiğimiz için **“İkinci yazımız”** metni gözüktükten sonra **“İlk yazımız”** metni

gözükeceğini tahmin etsek **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 geçebilmenin 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 fonksiyon yazısı”**’nı da beklemiş oluyoruz.

Anonim Goroutine fonksiyonları bu şekilde kullanabiliriz.

WaitGroup ile Asenkron İşlemleri Beklemek

Goroutine'leri Asenkron programlama yaparken kullanırız. Böylece aynı anda birden fazla işlem gerçekleştirebiliriz. Peki programımızın belirttiğimiz asenkron işlemleri bekleme gibi bir ihtiyacı olsaydı, ne yapmamız gerekirdi? Bu durumlarda WaitGroup'lardan faydalanabiliriz. Örneğin projemizde 3 adet asenkron işlem bulunuyorsa, WaitGroup'a 3 değerini ekleriz. Her asenkron işlem tamamlandığında WaitGroup -1 azalır ve sıfıra geldiğinde WaitGroup tamamlanmış olur. WaitGroup'u kullanmak için ise "sync" paketini projemize dahil ediyoruz. Kodlar üzerinde açıklamasını görelim.

```
{% code title="main.go" %}
```

```
package main
```

```
import (  
    "fmt"  
    "sync" //WaitGroup'u kullanmak için  
    "time" //bekleme işlemleri için  
)
```

```
/*  
* waitgroup nesnesini işaretçi olarak parametre veriyoruz.  
* işaretçi olarak vermemizin sebebi, programın bekleme işlemi için  
* asıl waitgroup nesnesini kontrol etmesidir.  
*/
```

```
func fonksiyon1(wg *sync.WaitGroup) {
```

```
    //fonksiyonun 2 sn beklemesini istiyoruz.  
    time.Sleep(2 * time.Second)  
    fmt.Println("Fonk1 tamamlandı")
```

```
    //wg.Done() fonksiyonu ile waitgroup nesnesini -1 azalttık.
```

```

    wg.Done()
}

//bu fonksiyonumuza da wg nesnesini işaretçi ile parametre
olarak verdik.
func fonksiyon2(wg *sync.WaitGroup) {
    //fonksiyonu 3 sn uyuttuk.
    time.Sleep(3 * time.Second)
    fmt.Println("Fonk2 tamamlandı")

    //-1 daha eksilttik.
    wg.Done()
}

func main() {
    /*
    * Öncelikle waitgroup'u kullanabilmek için bir waitgroup
    * nesnesi oluşturuyoruz.
    */
    var wg sync.WaitGroup

    /*
    * waitgroup'a 2 ekliyoruz. Yani 2 tane işlemden yanıt
    gelmesini
    * beklemesini istiyoruz. Aslında burada beklemeyecek.
    Sadece
    * işlem sayısını belirttik.
    */
    wg.Add(2)

    /*
    * fonksiyon1 ve fonksiyon2'ye oluşturduğumuz wg örneğinin
    * bellekteki adresinin veriyoruz.
    */
    go fonksiyon1(&wg)
    go fonksiyon2(&wg)
    fmt.Println("Merhaba Dünya!")

    /*
    * Burada wg.Wait() fonksiyonu ile asenkron işlemleri
    beklemesini
    * sağlıyoruz. yani waitgroup'un 0'a düşmesini bekliyoruz.
    * Eğer waitgroup olmadan yapsaydık. asenkron
    fonksiyonlarımızın tamamlanmasını
    * beklemeden program kendini sonlandırırdı.
    */
}

```



```
    */  
    wg.Wait()  
  
    //waitgroup tamamlandığında ekrana yazı bastıralım.  
    fmt.Println("WaitGroup'lar tamamlandı.")  
}  
  
{% endcode %}
```

Mutex ile Asenkron İşlem Sırası

Size konu başlığını şöyle açıklayayım. Örneğin bir banka uygulaması para çekme ve yatırma gibi özelliklere sahiptir. Programlama mantığında para yatırmak ve çekmek için mevcut para miktarını bilmemiz gerekir. Banka uygulamasının mantığı en basit derecede bu şekilde çalışır.

Banka hesabımızda asenkron işlem yapıldığını varsayalım. Yani bir hesaptan aynı anda birden fazla kullanıcı işlem yapıyor olsun.

Örneğin hesabımızda 100₺ olsun. Birinci kullanıcı 20₺ yatırsın. Aynı anda ikinci kullanıcı 50₺ çeksın. Bu iki kullanıcının kullandığı program işlem yapmaya başladığında önce para miktarını alıyor. Daha sonra yapılacak işleme göre ya ekleme ya da çıkarma işlemi yapıyor. Fakat birden fazla kullanıcı aynı anda bu işlemi yaparsa hesaptaki parada yanlışlık olacaktır.

Basit bir görsel ile inceleyelim.

İşlemlere aynı anda başlandığını varsayalım.

Para Miktarı: 100 TL



Örnek asenkron işlem

Bu işlemin sonuncunda hangi kullanıcının işlemi sonuncu olarak biterse para miktarı onun sonucu olur. Yani kullanıcı 2'nin işlemi kullanıcı 1'den sonra biterse yeni para miktarı 50₺ olur.

Bu gibi örneklerde asenkron işlemlere sıra verilmesi gerekir. Mutex tam olarak bu işi yapıyor. Bunun için bir Mutex nesnesi oluşturuyoruz. İşlemlerimizi bu nesne üzerinden yapıyoruz. Bu nesne aynı anda sadece bir işlemi gerçekleştiriyor. Bu yüzden sıra işlemi sağlıyor. Önce başlayan asenkron işlem ilk sırada oluyor. Tamamlanınca diğerine sıra geçiyor. Böyle düşündüğümüz zaman “bunun senkron programlamadan ne farkı var?” diyebilirsiniz. Farkı asenkron fonksiyonların içindeki istediğimiz kısımları senkron çalıştırmamız.

Örnek bir para yatırma-çekme uygulaması yazalım. İşlemin sağlıklı çalışması için, para miktarıyla aynı anda sadece bir kişi işlem yapabilmelidir.

```

package main

import (
    "fmt"
    "sync" // mutex'i kullanmak için
)
//global olarak mutex nesnesi oluşturalım.
var mt sync.Mutex

func paraÇek(bakiye *float64, çekilecekMiktar float64, wg
*sync.WaitGroup) {
    /*
    * mt isimli mutex'i bu işlem yapılırken kilitliyoruz.
    * bu sayede mt mutex'ini başka işlemler kullanamıyor.
    */
    mt.Lock()

    /*
    bu kısımda asenkron olmasını istemediğimiz işlemi yapalım.
    */
    *bakiye -= 15
    fmt.Printf("Yeni Bakiye: %.2f\n", *bakiye)

    /*
    * diğer işlemlerinde kullanabilmesi için mutex'i tekrardan
    açalım.
    * mt mutex açılınca diğer asenkron işlemdeki mt mutex'i
    çalışmaya başlar.
    */
    mt.Unlock()
    fmt.Println("Çekme işlemi tamamlandı.")

    /*
    * waitgroup ile işlemin tamamlandığını belirttik.
    * böylece wg havuzu 2'den 1'e düştü
    */
    wg.Done()
}

//bu fonksiyonda yukarıdaki ile aynı mantıkta
func paraYatır(bakiye *float64, yatırılacakMiktar float64, wg
*sync.WaitGroup) {
    mt.Lock()
    *bakiye += 65
    fmt.Printf("Yeni Bakiye: %.2f\n", *bakiye)
    mt.Unlock()
    fmt.Println("Yatırma işlemi tamamlandı.")
}

```

```

    wg.Done()
}

func main() {

    /*
    * asenkron işlemlerimizin, ana iş parçacığında tamamlanmasını
    * beklemek için waitgroup nesnesi oluşturalım
    */
    var wg sync.WaitGroup

    //2 fonksiyonu da bekleyeceğimiz için Add'e 2 yazalım
    wg.Add(2)

    //fonksiyonlarımızın kullanacağı bakiye değişkenimiz
    var bakiye float64 = 100
    fmt.Printf("İlk Bakiye: %.2f\n", bakiye)

    /*
    * paraÇek ve paraYatır fonksiyonlarımızı aynı anda başlatıyoruz.
    * hangisi daha önce başlarsa mutex sırasına ilk o girer. bu
    esnada diğer
    * fonksiyon mutex'in açılmasını bekler.
    */
    go paraÇek(&bakiye, 25, &wg)
    go paraYatır(&bakiye, 65, &wg)

    /*
    * ana iş parçacığı tamamlandığında asenkron çalışan
    fonksiyonları beklemez.
    * beklemediğinde de asenkron fonksiyonlar çalışmadan program
    sonlanır.
    * ana iş parçacığının asenkron işlemleri beklemesi için
    waitgroup sonucunun 0 olmasını bekleriz.
    * wg.Add(2) yazarak 2 adet wg.Done() fonksiyonu çalıştığında
    wg.Add(0) olur ve
    * wg.Wait() tamamlanır ve program başka işlemler yapılmıyor ise
    sonlanır.
    */
    wg.Wait()
}

```

Çıktımız aşağıdaki gibi olacaktır.

```

İlk Bakiye: 100.00
Yeni Bakiye: 165.00
Yatırma işlemi tamamlandı.

```

Yeni Bakiye: 150.00
Çekme işlemi tamamlandı.

Yukarıdaki çıktıya göre, paraYatır() fonksiyonu paraÇek()
fonkisiyonundan önce çalışmıştır.

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. Aşağıdaki **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. ($500 * x < 1600 \mid x = 3$) Haliyle de 3 kere ekrana çıktımızı bastırıyor. 1,6 saniye geçtikten sonra tekrar zamanlayıcımızı **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!

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.

init() Fonksiyonu (Ön Yükleme)

Golang'te 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() {
```

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

Çıktımız ise şöyle olacaktır.

Merhaba Dünya

İşte **init()** fonksiyonunun böyle bir güzellięi var. Benzer bir işlevi ancak pointers (işaretçiler) ile yapabiliriz.

```
package main  
import "fmt"  
var deęişken string = "Naber"  
func deęiştir(deęişken *string) {  
    *deęişken = "Merhaba Dünya"  
}  
func main() {  
    deęiştir(&deęişken)  
    fmt.Println(deęişken)  
}
```

O da gördüğünüz gibi uzun bir işlem oluyor.

Import (Kütüphane Ekleme) Yöntemleri

Bu yazıda sizlere Golang'ta paket import etmenin tüm yöntemlerini göstereceğim.

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 kullandığınız editör veya ide bu bölümü silebilir. import ederken _ **(alt tire)** koyarak bunun üstesinden gelebiliriz.

Dışa Aktarma (Exporting)

Golang'ta dışa aktarma 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(x, y int) int {  
    return x + y  
}
```

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

- Bir paket oluşturup başka bir paket içerisinde 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 aktarabiliriz. Örnek olarak:

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

Dışa aktarma olayı Golang'ta bu kadar basittir.

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 göreceğ 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 karakterleri metinsel bir ifade (string), dizgiyi formatlandırmak için kullanılır. Yani bir metinde değişken yerleri biçimlendirmeye yarar.

Format Karakteri	Açıklama
%T	Değişkenin tipini verir
%t	Boolean değeri verir
%d	Int (tamsayı) değeri verir
%b	Sayının binary (ikili) karşılığını verir
%c	Karakter değerini verir
%x	Sayının hexadecimal (onaltılı) karşılığını verir
%f	Float (ondalıklı) değeri verir
%s	String (dizgi-metin) değeri verir
%v	Değeri otomatik belirler

Hemen bir örnek yapalım.

```
package main

import "fmt"

func main() {
    isim := "Kaan"
    yaş := 23
    kilo := 71.3
    evli := false
```

```
    fmt.Printf("İsim: %s, Yaş: %d, Kilo: %f, Evli: %t", isim,
yaş, kilo, evli)
}
```

Yukarıdaki kodlara göre şöyle bir çıktı alacaksınız:

İsim: Kaan, Yaş: 23, Kilo: 71.300000, Evli: false

Kilo olarak girdiğimiz değer uzun olarak görüntülendi. Bunu değiştirmek için aşağıdaki yöntem uygulanır.

```
fmt.Printf("İsim: %s, Yaş: %d, Kilo: %.1f, Evli: %t", isim,
yaş, kilo, evli)
```

Yukarıdaki kodda farkedeceğiniz üzere kilo değişkeni için olan format karakterini `%.1f` olarak değiştirdik. Bu küsürlü sayılarda noktadan sonra 1 karakter gelebileceğini gösteriyor. Çıktımız: 71.3 olarak değişecektir.

İsim: Kaan, Yaş: 23, Kilo: 71.3, Evli: false

{% hint style="warning" %} Format karakterleri **Printf** ve **Scanf** gibi fonksiyonlarda kullanılabilir. Bu fonksiyonların ortak özellikleri adında **f** harfi olmasıdır. {% endhint %}

Kaçış Karakterleri ve Kullanım Alanları

Kaçış karakterleri de format karakterleri gibi metinlere etki eder. Kaçış karakterlerini kod yazma zamanında yapamadığımız işlemler için kullanırız.

Kaçış Karakteri

Açıklama

Komut satırında zil sesi çıkartır

Silme tuşu görevini görür

Merdiven metin yazar

Yeni satıra geçer

Return eder

Kaçış Karakteri

Tab tuşu gibi boşluk bırakır (4 boşluk)

Dikey boşluk bırakır

\

'

"

Açıklama

Ters-taksim yazar

Tek tırnak yazar

Çift tırnak yazar

Gelelim örneğimize:

```
fmt.Print("Bir\nİki\tÜç\\Dört")
```

Çıktımız şöyle olacaktır:

```
Bir
İki Üç
```

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)
}
```

Sprintf

Sprintf fonksiyonu fmt paketine dahil bir fonksiyondur. Bu fonksiyon değişkenlere formatlı atama yapmamıza yardımcı olur. Örneğimizi görelim:

```
package main

import (
    "fmt"
)

func main() {
    isim := "Kaan"

    isimTip := fmt.Sprintf("isim değişkeni %T tipindedir.",
isim)

    fmt.Println(isimTip)
}
```

Golang'te Kullanıcıdan Giriş Alma

Golang'te 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')
```

{% hint style="info" %} **Scan** komutu ile kelime alamadığınızda **Reader** ile deneyebilirsiniz. {% endhint %}

Testing (Test Etme)

Hücrelerin vücudadaki 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 görüldüğünü 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
```

Bu yazımızda Golang’de test işleminin nasıl yapıldığını gördük. Mantığını daha iyi kavramak için bir proje üzerinde

gerekli olduđu yerde kullanmamız gerekir.

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
```

İstersek biz de kritik bir bilginin nil girilmesi gibi durumlarda programı durdurabiliriz. Bunun için **panic()** fonksiyonunu kullanacağız. Hemen bir örnek yapalım.

```
package main

import (
    "fmt"
```

```

)

func TamIsim(Ad *string, Soyad *string) {
    if Ad == nil {
        panic("Ad nil olamaz")
    }
    if Soyad == nil {
        panic("Soyad nil olamaz")
    }
    fmt.Printf("%s %s\n", *Ad, *Soyad)
    fmt.Println("TamIsim fonksiyonu bitti")
}

func main() {
    Ad := "Yusuf"
    TamIsim(&Ad, nil)
    fmt.Println("Ana fonksiyon da bitti")
}

```

Çıktımız burada:

```

panic: Soyad nil olamaz
goroutine 1 [running]:
main.TamIsim(0xc00007df30, 0x0)
/Users/Y/Desktop/main.go:12 +0x19a
main.main()
/Users/Y/Desktop/main.go:20 +0x65
exit status 2

```

Burada **Soyad** değişkeni tanımsız olduğu için programımız durdu. Aynı şekilde **recover()** fonksiyonu ile **panic()** fonksiyonundan gelen veriyi alabilir, ana fonksiyonumuzun kapanmasına da engel olabiliriz. Bunun için de bir örnek yapalım.

```

package main

import (
    "fmt"
)

```

```
func TamIsim(Ad *string, Soyad *string) {
    if Ad == nil {
        panic("Ad nil olamaz")
    }
    if Soyad == nil {
        panic("Soyad nil olamaz")
    }
    fmt.Printf("%s %s\n", *Ad, *Soyad)
    fmt.Println("TamIsim fonksiyonu bitti")
}

func main() {
    Ad := "Yusuf"
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Panik Yok : ", r)
        }
    }()
    TamIsim(&Ad, nil)
    fmt.Println("Ana fonksiyon da bitti")
}
```

Çıktımız burada :

Panik Yok : Soyad nil olamaz

Strings

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

Strings.Contains() Fonksiyonu

Contains() fonksiyonu ile istediğimiz bir string değerinin içerisinde istediğimiz bir **string** değer 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**. **Contains()** 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 kaçınıcı 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 değerın tüm harflerini büyük harf yapar.

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

Çıktımız **"MERHABA DÜNYA"** olacaktır.

Strings.ToLower() Fonksiyonu

ToLower() fonksiyonu içerisine girilen string değerın 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,  
"ıİÜÖÇ"))
```

Çıktımız “**İİÜÖÇ**” olacaktır.

Strings.ToLowerSpecial() Fonksiyonu

ToUpperCaseSpecial() fonksiyonu ile aynı şekilde çalışır ;fakat harfleri belirlediğiniz karakter kodlamasına göre küçültür.

Örnek kullanımı:

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

Çıktımız “**ııüöç**” olacaktır.

os/exec (Komut Satırına Erişim)

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 gereken 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 Wink

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")
    cmd0kuyucu, hata := cmd.StdoutPipe()
    if hata != nil {
        fmt.Fprintln(os.Stderr, "Çıktı okunurken hata oluştu:",
hata)
        os.Exit(1)
    }
    //çıktı okuyucusunun tanımlanması
    çıktı := bufio.NewScanner(cmd0kuyucu)
    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
oluştı:", 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. **çikti** adında değişkenimizi oluşturuyoruz. Bu değişkenimiz **cmdOkuyucu** değişkenini taramaya yarayacak. Hemen aşağısında goroutine fonksiyonumuzda **çikti.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 düşünüyorum.

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", "version")
    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.

Komut Satırı Argümanları (Args)

Golang ile programlarımızın komut satırı üzerinden argümanlar ile çalışmasını sağlayabiliriz. İşte paketimiz:

```
import "os"
```

os paketimizdeki Args fonksiyonu bize string dizi sunar.
****Bir örnek görelim.

```
package main
```

```
import (  
    "fmt"  
    "os"  
)
```

```
func main() {  
    for i, arg := range os.Args {  
        fmt.Println(i, "=", arg)  
    }  
}
```

for-range ile os.Args'ın uzunluğu kadar işlem yapıyoruz ve içerisindekileri indeksi ile ekrana bastırıyoruz. Şöyle bir çıktımız oluyor:

```
./main naber nasılsın
```

```
0 = ./main  
1 = naber  
2 = nasılsın
```

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**” argümanı ile 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**” argümanı ile gireceğimizi belirttik. Varsayılan değeri **1881** olacak ve açıklama bölümünde “**sayı tipinde**” yazacak.

mantiksal isminde **bool** tipinde bir flag oluşturduk. **flag.Bool()** fonksiyonunda “**-mantiksal**” argümanı ile 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 (default false)  
-sayi int  
sayı tipinde (default 1881)
```

os/signal

os/signal paketi gelen sinyallere erişim sağlar. Genellikle Unix-benzeri sistemlerde kullanılır. **Windows** ve **Plan9**'da kullanımı farklıdır.

Sinyal Türleri

SIGKILL ve **SIGSTOP** sinyalleri bir program tarafından yakalanmayabilir ve bu nedenle bu paketten etkilenemez.

Senkron sinyaller, program yürütmedeki hatalarla tetiklenen sinyallerdir: **SIGBUS**, **SIGFPE** ve **SIGSEGV**. Bunlar, `os.Process.Kill` veya **kill** programı veya benzer bir mekanizma kullanılarak gönderildiklerinde değil, yalnızca program yürütülmesinden kaynaklandığında eşzamanlı olarak kabul edilir. Genel olarak, aşağıda tartışılanlar dışında, Go programları eşzamanlı bir sinyali çalışma zamanı paniğine dönüştürecektir.

Kalan sinyaller asenkron sinyallerdir. Program hataları tarafından tetiklenmezler, bunun yerine çekirdekten veya başka bir programdan gönderilirler.

Asenkron sinyallerden, **SIGHUP** sinyali, bir program kontrol terminalini kaybettiğinde gönderilir. **SIGINT** sinyali, kontrol terminalindeki kullanıcı, varsayılan olarak ^ C (Kontrol-C) olan kesme karakterine bastığında gönderilir. **SIGQUIT** sinyali, kontrol terminalindeki kullanıcı varsayılan olarak ^ (Kontrol-Ters Taksim) olan çıkış karakterine bastığında gönderilir. Genel olarak, bir programın ^ C'ye basarak çıkmasına neden olabilir ve ^ tuşuna basarak bir yığın dökümü ile çıkmasına neden olabilirsiniz.

Go Programlarında Sinyallerin Varsayılan Davranışı

Varsayılan olarak, senkronize bir sinyal çalışma zamanı paniğine dönüştürülür. **SIGHUP**, **SIGINT** veya **SIGTERM** sinyali programın çıkmasına neden olur. **SIGQUIT**, **SIGILL**, **SIGTRAP**, **SIGABRT**, **SIGSTKFLT**, **SIGEMT** veya **SIGSYS** sinyali, programın yığın dökümü ile çıkmasına neden olur. Bir **SIGTSTP**, **SIGTTIN** veya **SIGTTOU** sinyali sistem varsayılan davranışını alır (bu sinyaller kabuk tarafından iş kontrolü için kullanılır). **SIGPROF** sinyali, `runtime.CPUProfile`'i uygulamak için doğrudan Go çalışma zamanı tarafından işlenir. Diğer sinyaller yakalanacak ancak herhangi bir işlem yapılmayacaktır.

Go programı, **SIGHUP** veya **SIGINT** göz ardı edilerek başlatılırsa (sinyal işleyici **SIG_IGN**'a ayarlı), bunlar ihmal edilmiş olarak kalacaktır.

Go programı boş olmayan bir sinyal maskesi ile başlatılırsa, bu genellikle kabul edilir. Bununla birlikte, bazı sinyaller açıkça engellenmiştir: eşzamanlı sinyaller, **SIGILL**, **SIGTRAP**, **SIGSTKFLT**, **SIGCHLD**, **SIGPROF** ve **GNU/Linux'ta 32 (SIGCANCEL) ve 33 (SIGSETXID) (SIGCANCEL ve SIGSETXID)** sinyalleri **glibc** tarafından dahili olarak kullanılır. `os.Exec` veya `os/exec` paketi tarafından başlatılan alt işlemler, değiştirilmiş sinyal maskesini miras alır.

Go Programlarında Sinyallerin Davranışını Değiştirme

Bu paketteki işlevler, bir programın Go programlarının sinyalleri işleme şeklini değiştirmesine izin verir.

Notify, belirli bir eşzamansız sinyal kümesi için varsayılan davranışı devre dışı bırakır ve bunun yerine bunları bir veya daha fazla kayıtlı kanal üzerinden iletir. Özellikle, **SIGHUP**, **SIGINT**, **SIGQUIT**, **SIGABRT** ve **SIGTERM** sinyalleri için geçerlidir. Bu aynı zamanda iş kontrol sinyalleri **SIGTSTP**, **SIGTTIN** ve **SIGTTOU** için de geçerlidir ve bu durumda sistem varsayılan davranışı oluşmaz. Aynı zamanda, başka şekilde hiçbir eyleme neden olmayan bazı sinyaller için de geçerlidir: **SIGUSR1**, **SIGUSR2**, **SIGPIPE**, **SIGALRM**, **SIGCHLD**, **SIGCONT**, **SIGURG**, **SIGXCPU**, **SIGXFSZ**, **SIGVTALRM**, **SIGWINCH**, **SIGIO**, **SIGPWR**, **SIGSIGTHEZW**, **SIGTHAW**, **SIGLOST**, **SIGXRES**, **SIGJVM1**, **SIGJVM2** ve sistemde kullanılan gerçek zamanlı sinyaller. Bu sinyallerin tümünün tüm sistemlerde mevcut olmadığını unutmayın.

Program **SIGHUP** veya **SIGINT** göz ardı edilerek başlatılmışsa ve her iki sinyal için de **Notify** çağrılırsa, bu sinyal için bir sinyal işleyici kurulacak ve artık göz ardı edilmeyecektir. Daha sonra bu sinyal için **Reset** veya **Ignore** çağrılırsa veya o sinyal için **Notify**'ye iletilen tüm kanallarda **Stop** çağrılırsa, sinyal bir kez daha yok sayılır. **Reset**, sinyal için sistemin varsayılan davranışını geri yüklerken, **Ignore**, sistemin sinyali tamamen yok saymasına neden olur.

Program boş olmayan bir sinyal maskesi ile başlatılırsa, bazı sinyallerin blokajı yukarıda açıklandığı gibi açıkça kaldırılacaktır. Engellenen bir sinyal için **Notify** çağrılırsa, engellemesi kaldırılır. Daha sonra bu sinyal için **Reset** çağrılırsa veya bu sinyal için **Notify**'ye iletilen tüm kanallarda **Stop** çağrılırsa, sinyal bir kez daha engellenecektir.

Windows

Windows'ta a ^ C (Control-C) veya ^ BREAK (Control-Break) normalde programın çıkmasına neden olur. `os.Interrupt` için **Notify** çağrılırsa, ^ C veya ^ BREAK `os.Interrupt`'ın kanala gönderilmesine neden olur ve program çıkmaz. **Notify**'ye geçen tüm kanallarda **Reset** çağrılırsa veya **Stop** çağrılırsa, varsayılan davranış geri yüklenir.

Plan9

Plan 9'da, sinyaller bir dizge olan `syscall.Note` türüne sahiptir. **Notify** ile bir sistem çağrısı çağırmak, bu dize bir not olarak gönderildiğinde bu değerin kanala gönderilmesine neden olur.

Örnek Uygulama

```
{% code title="main.go" %}

package main

import (
    "fmt"
    "os"
    "os/signal"
    "time"
)

func main() {
    //os.Signal tipinde değer taşıyan bir kanal oluşturduk
    signalKanalı := make(chan os.Signal, 1)

    /*
    * Ana iş parçacığı sonlanmaması için bir kanal
    * oluşturalım.
    */
    programBitti := make(chan bool)

    /*
    * os.Interrupt sinyali ile programın sonlanması
    * yerine sinyali signalKanalı'na yönlendirelim.
```

```

    */
    signal.Notify(signalKanalı, os.Interrupt)

    /*
    * asenkron olarak signalKanalı'nı dinleyelim. Sinyal
    * geldiğinde yani CTRL + C'ye basıldığında for döngüsü
    * içerisindeki kodlar çalışacak.
    */
    go func() {
        for range signalKanalı {
            fmt.Println("Kontrol + C 'ye basıldı")

            //5 sn bekleyelim.
            time.Sleep(time.Second * 5)

            /*
            * Burada bekleyerek size programın CTRL + C'ye
            * basıldığında kapanmadığını gösteriyorum :)
            */
            fmt.Println("bitti")

            /*
            * Kanala değer göndererek ana iş parçacığındaki
            * programBitti kanalının bekleyişine son verelim.
            */
            programBitti <- true
        }

    }()

    //Ana iş parçacığı sonlanmasın diye kanal bekleyelim
    <-programBitti
}

{% endcode %}

```

Örnek: Tüm Sinyalleri Yakalamak

```
package main
```

```
import (
    "fmt"
    "os"

```



```
    "os/signal"  
)  
  
func main() {  
    //Sinyallerin gönderileceği kanalı oluşturalım.  
    kanal1 := make(chan os.Signal, 1)  
  
    //Gelen sinyalleri kanal1'e yönlendirelim  
    signal.Notify(kanal1)  
  
    // kanal1'e sinyal gelene kadar programı bekletelim.  
    sinyalTürü := <-kanal1  
    fmt.Println("Sinyal Türü:", sinyalTürü)  
}
```

Sort (Sıralama)

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)
    yazisiralali := sort.StringsAreSorted(yazilar)
    fmt.Println("Yazılar Sıralandı mı?: ", yazisiralali)
    sayisiralali := sort.IntsAreSorted(sayilar)
    fmt.Println("Sayılar Sıralandı mı?:", sayisiralali)
}
```

Gelelim açıklamasına;

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

yazilar isminde içerisinde rastgele harflerden 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 (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 kullanımlarını 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
```

Log (Kayıt)

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.

```
{% page-ref page="../boeluem-5/init-fonksiyonu-oen-yuekleme.md" %}
```

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 **log.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 geçildiğini 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.

Paket (Kütüphane) Yazmak

Bu bölümde Go üzerinde nasıl kendi paketimizi (kütüphanemizi) oluşturacağımıza bakacağız.

Bir Paketin Özellikleri

- İçerisinde .go dosyaları bulunan bir klasördür.
- Diğer projeler tarafından içe aktarılabilir.
- Dışa aktarılabilen veya aktarılamayan veriler içerir.
- Açık kaynaktır.
- `main()` fonksiyonu içermez.
- `package main` değildir.

Paket oluştururken dikkat etmemiz gereken prensipler vardır. Bunlar şık ve basit kod yazımı, dışarıdan kullanımı basit, mümkün olduğunca diğer paketlere bağımsız olmasıdır. Bu prensiplere dikkat ederek daha iyi bir paket yazabilirsiniz.

Proje Klasöründe Yerel Kütüphane Oluşturma

Öncelikle aşağıdaki gibi bir dosya düzenimiz olduğunu varsayalım.



Proje Klasörümüzün Yapısı

Yukarıdaki gibi paketim klasörü içerisinde paketim.go dosyamız olsun.

paketim.go dosyamızın içi aşağıdaki gibi olsun.

```
package paketim

import "fmt"

func Yaz() {
    fmt.Println("yazdım!")
}
```

package paketim ile paketimizin ismini belirledik. Bu isim paket klasörümüz ile aynı olmalıdır. Daha sonra projemizde kullanabilmemiz için dışa aktarılmış şekilde Yaz() fonksiyonu oluşturduk. Bu fonksiyonun ne işe yaradığı zaten belli.

main.go dosyamız ise aşağıdaki gibi olsun.

```
package main

import p "./paketim"

func main() {
    p.Yaz()
}
```

`import p "./paketim"` yazarak özel paketimizin yerel konumunu belirterek, `p` lakabı (alias) ile çağırdık.

`Yaz()` fonksiyonumuzu ise `p.Yaz()` şeklinde kullandık.

Git Sisteminde Kütüphane Paylaşımı

Oluşturduğumuz kütüphaneyi Github, Gitlab, Bitbucket vb. sitelerde barındırarak diğer geliştiricilerinde kütüphanelerinizi faydalanmasını sağlayabilirsiniz.

Bunun için kütüphanenizin isminde bir repo oluşturup, içerisinde Go dosyalarınızı yükleyin. Daha sonra `go get github.com/id/repoismi` şeklinde projenize import edebilirsiniz.

Regex (Kurallı İfadeler)

Regular Expressions (Regex), modern programlama dillerinin neredeyse hepsinde bulunan metinsel ifadelerinizin yapısını kontrol etmenizi sağlayan bir pakettir.

Bu paket sayesinde yazdığımız programda ifadelerin uygunluğunu kontrol edebilir, işimize yarayacak ifade/ifadeleri daha kolay ayırabilir ve giriş yapılan ifadeleri uygun bir düzene koyabiliriz.

Örneğimizi görelim

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    var isimKontrol = regexp.MustCompile(`^[a-z]+[0-9]$`)

    fmt.Println(isimKontrol.MatchString("kaan10")) //false
    fmt.Println(isimKontrol.MatchString("emir6"))  //true
    fmt.Println(isimKontrol.MatchString("gokhan")) //false
    fmt.Println(isimKontrol.MatchString("Altan2")) //false
    fmt.Println(isimKontrol.MatchString("8erkay")) //false
}
```

Öncelikle yukarıdaki işlemleri yapabilmemiz için, projemizde "regexp" paketini çağırmamız gerekiyor.

isimKontrol adında bir değişken oluşturduk ve bu değişkenimizde MustCompile() fonksiyonu ile metinsel ifademizin kurallarını belirledik.

Belirlediğimiz kural ise dizgimizin küçük harflerle a'dan z'ye kadar ve ek olarak 0'dan 9'a kadar **"rakamsal"** ifade alabileceğidir. Bu kurala uyulması için metinsel ifade ne eksik ne fazla hiçbir şey olmaması gerekir.

Kuralımızın hemen aşağısındaki örnekte ise "kaan10" ifadesinde sadece 0-9 arası rakam (tek haneli) olması gerektiği için false çıktısını verdi.

"emir6" ifade belirttiğimiz kurala uygun bir ifade olduğu için true çıktısını verdi.

"gokhan" ifadesi içerisinde sayı barındırmadığı için false çıktısını verdi.

"Altan2" ifadesi büyük harf ile başladığı için false çıktısını verdi.

"8erkay" ifadesi ise rakam sonda olması gerekirken başta olduğu için false çıktısını verdi.

Bu yazıda regexp nasıl yazılırdan ziyade Go'da regexp'in kullanımını anlatmak istediğimden bu kısmı kısa tutuyorum. Regexp'in nasıl yazıldığına göz atmak isteyenler için aşağıda linkini paylaşıyorum. Buyursunlar: [Regex - Regular Expressions Nedir?](#)

Bu güzel anlatım için [Ceyhun Enki Aksan](#)'a ayrıca teşekkür ederim.

Çapraz Platform Dosya Yolları

Bir işletim sisteminde dosyanın veya dizinin yolunu belirtmek için taksim veya ters-taksim işaretleri kullanırız. Fakat yazağımız program çapraz-platformsa bu durumda ne yapmamız gerekir?

Ya kendimiz bunun için bir fonksiyon oluşturacağız ya da kısa yoldan `os.PathSeparator`'ı kullanabiliriz.

Hemen örneğimizi görelim:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    s := string(os.PathSeparator)
    yol := "dosyalar" + s + "muzikler"
    fmt.Println(yol)
}
```

{% hint style="info" %} Her seferinde `string(os.PathSeparator)` yazmamak için `s` değişkenine atayarak kısalttık. {% endhint %}

Windows için çıktımız:

dosyalar

Unix-Like için çıktımız:

dosyalar/muzikler

İşletim Sistemini Görme

Go programının çalıştığı işletim sistemi görmek için aşağıdaki kodları yazabilirsiniz.

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    if r := runtime.GOOS; r == "windows" {
        fmt.Println("Windows için yönetici olarak çalıştırın.")
    } else if r == "linux" {
        fmt.Println("Linux için sudo komutu ile çalıştırın.")
    } else {
        fmt.Println("Geçersiz işletim sistemi!")
    }
}
```

GNU/Linux kullandığım için çıktım aşağıdaki gibi olacaktır.

Linux için sudo komutu ile çalıştırın.

Dosya Varlığı Kontrolü

Go programımızda kullanacağımız bir dosyanın varlığını `os` paketi ile kontrol edebiliriz. Örnek programımızı görelim:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    if d := "dosya.txt"; dosyaVarmı(d) {
        fmt.Println(d, "bulunuyor")
    } else {
        fmt.Println(d, "bulunmuyor!")
    }
}

func dosyaVarmı(isim string) bool {
    bilgi, hata := os.Stat(isim)
    if os.IsNotExist(hata) {
        return false
    }
    return !bilgi.IsDir()
}
```

Gelelim açıklmasına:

Dosya işlemleri yapabilmek için `os` paketini import ettik. `if-else` akışında geçici değişken olarak `d` değişkenine `"dosya.txt"` atayarak kontrol edilecek dosyamızın ismini belirledik.

Bu akışta `dosyaVarmı` fonksiyonunda `true` değer dönerse `dosya.txt` bulunuyor olarak çıktı almamız gerekir.

`dosyaVarmi` fonksiyonunu incelediğimizde `bilgi` ve `hata` değişkenlerine `os.Stat` ile dosyanın bilgilerini çektik. `hata` değişkeni `false` döndürürse fonksiyonun `false` döndürmesini istedik. Aynı şekilde `bilgi.IsDir()` ile dosya değil de bir dizinse `false` döndürmesini istedik.

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))
}
```

{% hint style="info" %} Okuma işlemi **byte** tipinde yapıldığı için **string()** fonksiyonu ile byte tipini string tipine dönüştürüyoruz. {% endhint %}

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)
}
```

{% hint style="info" %} String tipini dosyaya yazdırmamız için önce byte tipine çevirmemiz gerekir. {% endhint %}

Dosya yazdırma işleminde aynı isimde dosya varsa üzerine yazar.

Bir Dizindeki Dosya ve Klasörleri Sıralama

Golang üzerinde adresini belirlediğimiz bir dizindeki dosya ve klasörleri listelemeyi göreceğiz. Örneğimize geçelim:

```
package main
import (
    "fmt"
    "os"
)
func diziniOku(d string) {
    dizin, err := os.Open(d)
    if err != nil {
        fmt.Println("Dizin bulunamadı!")
        os.Exit(1)
    }
    defer dizin.Close()
    liste, _ := dizin.Readdirnames(0) // Açıklamada okuyun
    for _, isim := range liste {
        fmt.Println(isim)
    }
    fmt.Printf("Toplamda %d içerik bulundu.\n", len(liste))
}
func main() {
    diziniOku(".")
}
```

Yukarıdaki kodlarımızın açıklamasını görelim:

Öncelikle **“os”** paketimizi içe aktarıyoruz. **diziniOku()** fonksiyonumuzun içerisinde **dizin** adında değişken oluşturduk ve bu değişkende fonksiyonumuza **d** argümanı ile gelecek olan dizinimizi açtık. Eğer bir hata ile karşılaşırsak diye hata yakalama işlemi yaptık. Daha sonra **dizin** değişkenimizi **defer** ile kapattık. **liste** adında değişken oluşturduk. Bu değişkenimizin içerisine **dizin.Readdirnames(0)** diyerek tüm dosya ve

klasörleri bu değişkenimizin içerisine attık. Burada sıfır kullanmamızın sebebi tüm dosya ve klasörleri okuyabilmek içindir.

Hemen aşağısında **for** ve **range** ile **liste** değişkenimizdeki dosya ve klasör isimlerini isim değişkenimize bastırmak istedik. Her dosya ve klasör ayrı ayrı isim değişkenimize atandı ve ekrana bastırılmış oldu.

Daha sonra **diziniOku()** fonksiyonumuzun en altında **len(liste)** ile dosya sayımızı öğrenerek ekrana bastırdık.

main() fonksiyonumuzda ise **diziniOku(".")** diyerek nokta ile bulunduğumuz dizini okuttuk.

XML Parsing (Ayrıştırma)

Bu yazımıza 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)  
}
```

JSON Parsing (Ayrıştırma)

Yazıya başlamadan önce bu konuyu yazdığı için **Latif Uluman**'a ([Twitter](#)) teşekkürlerimi sunarım.

Bugünkü yazımızda *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)**

Evet 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 hiçbirisi 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}

Evet arkadaşlar görüldüğü gibi kodumuz çalıştı. Şimdi kısaca açıklayalım programımızı:

7-11 satırlarda kendi **“kişi”** tipimizi oluşturduk. **13-17** satırlarda 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. **19-22** satırlarda hatayı kontrol ettik. Ve son olarak da hatalı değilse ekrana bastık. Tabii bizim datamız []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(alı, "", "    ")
    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 parametre 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. Evet **“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

Evet arkadaşlar 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 gerçekleştirmemiz 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.Evet arkadaşlar bu yazımızda nasıl bir go verisini json verisine dönüştürüp kullanacağımızı ya da tam tersi json verisini go verisine dönüştüreceğimizi gördük.

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")
}

```

Web Scraper (goquery)

Bu yazıda Go dilinde nasıl basitçe web scraper yapacağımıza bakacağız.

Web Scraper Nedir?

Web Scraper bir web sayfasındaki elementleri işleyen araçtır.

Örnek uygulama:

blog.golang.org sitesindeki blog başlıklarını listeleyen Go programının yazılması.

```
package main

import (
    "fmt"
    "log"
    "net/http"

    "github.com/PuerkitoBio/goquery"
)

func main() {
    blogBasliklari, err := baslikCek("https://blog.golang.org")
    if err != nil {
        log.Println(err)
    }
    fmt.Println("Başlıklar:")
    fmt.Printf(blogBasliklari)
}

// URL adresinden blog başlıklarını çekecek fonksiyon
func baslikCek(url string) (string, error) {

    // HTML'i çek
    resp, err := http.Get(url)
```

```

    if err != nil {
        return "", err
    }

    // goquery dökümanına çevir
    doc, err := goquery.NewDocumentFromReader(resp.Body)
    if err != nil {
        return "", err
    }

    // liste oluştur
    basliklar := ""
    doc.Find(".title a").Each(func(i int, s *goquery.Selection)
{
        basliklar += "- " + s.Text() + "\n"
    })
    return basliklar, nil
}

```

Açıklaması:

goquery kütüphanesini bilgisayarımıza indiriyoruz.

`go get github.com/PuerkitoBio/goquery`

baslikCek fonksiyonuna URL adresini girdik. Zaten bu fonksiyonu da bi oluşturduk. Hata kontrolü yaptıktan sonra başlıkları yazdırdık.

baslikCek fonksiyonuna baktığımızda;
İlk önce url adresini, yani içindeki elementleri, çektik.
goquery dökümanına çevirdik. Burada dikkat edilmesi gereken nokta, resp değişkeni bizim çektiğimiz url adresidir. Daha sonra liste olarak oluşturduk. Liste oluşturma işleminde .title sınıfına ait ve a etiketinde olan elementleri sıralamasını istedik. Element seçim işlemi jQuery selector mantığında çalışır.

Çıktımız:

Başlıklar:

- [Announcing the 2019 Go Developer Survey](#)
- [Go.dev: a new hub for Go developers](#)
- [Go Turns 10](#)
- [Go Modules: v2 and Beyond](#)
- [Working with Errors in Go 1.13](#)

chromedp (Web Driver)

Chromedp paketi, harici bağımlılıklar (Selenium veya PhantomJS gibi) olmadan Go'da **Chrome DevTools Protokolünü** destekleyen tarayıcıları çalıştırmanın daha hızlı ve daha basit bir yoludur. Harici bağımlılık yoktur derken, tabi ki sisteminizde Google Chrome'un yüklü olması gerekiyor. Chromedp'ye headless modu gerektiği için minimum Chrome sürümünüz 59 olması gerekiyor.

Paketi yüklemek için:

```
go get -u github.com/chromedp/chromedp
```

Örnek.1

```
package main

import (
    "context"
    "log"

    "github.com/chromedp/chromedp"
)

func main() {
    //chrome örneği oluşturalım
    ctx, cancel := chromedp.NewExecAllocator(
        context.Background(),
        append(
            chromedp.DefaultExecAllocatorOptions[:],
            chromedp.Flag("headless", false),
        )...,
    )
    //headless: false ayarlayarak pencerenin görünmesini
    istedik
```

```
//chrome nesnesini defer ile kapatmayı unutmuyoruz
defer cancel()

//yeni durum oluşturunuz
ctx, cancel = chromedp.NewContext(ctx)

//aynı şekilde defer ile penceremizide kapatıyoruz
defer cancel()

//Twitter isminin kaydedileceği değişkeni oluşturalım
var twitterName string

//chromedp.Run() içerisinde tarayıcıda yapılacak işlemleri
yazıyoruz.
err := chromedp.Run(ctx, //önce durumu (hangi pencere)
olacağını belirtiyoruz

    //tarayıcının gitmesini istediğimiz adresi yazalım
    chromedp.Navigate(`https://kaanksc.com/posts/webview-
statik-uygulama-ornegi3/`),

    //css seçici ile belirttiğimiz elementin yüklenmesini
bekleyelim
    chromedp.WaitVisible(`.single__contents > p:nth-
child(16) > a:nth-child(1)`, chromedp.ByQuery),

    //Tıklanılacak nesneyi yine css seçici ile belirtelim
    chromedp.Click(`.single__contents > p:nth-child(16) >
a:nth-child(1)`, chromedp.ByQuery),
    //Bu işlemden sonra twitter'a gidecek

    //Twitter profilinde adın gösterildiği yeri css seçici
ile beklemesini istedik
    chromedp.WaitVisible(`div.r-1b6ydlw:nth-child(1) >
span:nth-child(1)`, chromedp.ByQuery),

    //belirttiğimiz css seçicisi ile elementin içindeki
yazıyı twitterName değişkenine atayalım
    chromedp.Text(`div.r-1b6ydlw:nth-child(1) > span:nth-
child(1)`, &twitterName),

    //burdan sonra tarayıcı penceresi kapanacak
)

//hata kontrolü yapalım
```



```

    if err != nil {
        log.Fatal(err)
    }

    //son olarak twitterName içindeki değişkeni ekrana
    bastıralım
    log.Printf("Twitter İsim:%s\n", twitterName)
}

```

Yukarıdaki örnekte yeni chrome penceresi oluşturma, tıklama, elementin yüklenmesini bekleme, element içindeki yazıyı alma ve adrese gitme gibi işlemlerin nasıl yapıldığını gördük.

Örnek.2

Go Playground linkinden Go kodlarını çeken bir uygulama yazalım.

```

package main

import (
    "context"
    "log"

    "github.com/chromedp/chromedp"
)

func main() {
    //chrome örneği oluşturalım
    ctx, cancel := chromedp.NewExecAllocator(
        context.Background(),
        append(
            chromedp.DefaultExecAllocatorOptions[:],
            chromedp.Flag("headless", true), //Bu sefer
            headless çalışmasını istedik
            //yani chrome penceresi açılmayacak
        )...,
    )

    //chrome nesnesini defer ile kapatmayı unutmuyoruz
    defer cancel()
}

```

```

//yeni durum oluşturunuz
ctx, cancel = chromedp.NewContext(ctx)

//aynı şekilde defer ile penceremizde kapatıyoruz
defer cancel()

//go kodlarının kaydedileceği değişkeni oluşturalım
var goKodu string

//chromedp.Run() içerisinde tarayıcıda yapılacak işlemleri
yazıyoruz.
err := chromedp.Run(ctx, //önce durumu (hangi pencere)
olacağını belirtiyoruz

//tarayıcının gitmesini istediğimiz adresi yazalım
chromedp.Navigate(`https://play.golang.org/p/a_SoTzENmV7`),

//tarayıcının yüklenmesini bekleyeceği elementi css
seçici ile yazıyoruz
chromedp.WaitVisible(`#code`, chromedp.ByQuery),

//textContent ile yazı alanı içeriğini çekebiliriz.
chromedp.TextContent(`#code`, &goKodu,
chromedp.ByQuery),
)

if err != nil {
    log.Fatal(err)
}

//son olarak go kodlarını ekrana bastıralım
log.Printf("Go Kodu:\n%s", goKodu)
}

```

Yukarıdaki örnekte headless modda çalışmayı ve yazı kutusu (input veya textarea) içindeki yazıları almayı öğrendik.

Daha fazla bilgi için <https://github.com/chromedp/chromedp>, daha fazla örnek için <https://github.com/chromedp/examples> adresine bakabilirsiniz.

MySQL

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 kullanıcı şifre 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,
    kullanıcı TEXT NOT NULL,
    şifre 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,
        kullanıcı TEXT NOT NULL,
        şifre TEXT NOT NULL,
        tarih DATETIME,
```

```
        PRIMARY KEY (id)
    );`
//Sorguyu çalıştırma
_, err := db.Exec(sorgu)
```

Bu işlemle birlikte MySQL veritabanımızda **kullanıcılar** 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 kullanıcı struct {
    id      int
    kullanıcı string
    sifre   string
    tarih   time.Time
}
tablo, _ := db.Query(`SELECT id, kullanıcı, sifre, tarih FROM
kullanıcılar`)
defer rows.Close()
var kullanıcılar []kullanıcı
for tablo.Next() {
    var k kullanıcı
    err := tablo.Scan(&k.id, &k.kullanıcı, &k.sifre, &k.tarih)
    kontrol(err)
    kullanıcılar = append(kullanıcılar, k)
}
err := tablo.Err()
kontrol(err)
```

Bu işlemin sonucunda kullanıcılar dizimiz şu şekilde olacaktır.

```
kullanıcılar {
    kullanıcı {
        id:      1,
        kullanıcı: "ahmet",
        sifre:   "1234",
        tarih:   time.Time{wall: 0x0, ext: 63701044325, loc:
(*time.Location)(nil)},
    },
    kullanıcı {
        id:      2,
        kullanıcı: "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
        kullanıcı := "kaan"
        şifre := "1234"
        tarih := time.Now()
        sonuc, err := db.Exec(`INSERT INTO kullanicilar
(kullanici, şifre, tarih) VALUES (?, ?, ?)`, username,
password, createdAt)
        kontrol(err)
        eklenenid, err := sonuc.LastInsertId()
        fmt.Println(eklenenid)
    }
    { // İstenilen kaydı sorgulama
        var (
            id          int
            kullanıcı string
            şifre        string
            tarih        time.Time
            sorguid      int = 1
        )
        sorgu := "SELECT id, kullanıcı, şifre, tarih FROM
kullanici WHERE id = ?"
        err := vt.QueryRow(sorgu, sorguid).Scan(&id,
&kullanici, &şifre, &tarih)
        kontrol(err)
        fmt.Println(id, kullanıcı, şifre, tarih)
    }
    { // Tüm kayıtları sorgula
        type kullanıcı struct {
            id          int
            kullanıcı string
            şifre        string
            tarih        time.Time
        }
        tablo, err := vt.Query(`SELECT id, kullanıcı, şifre,
tarih FROM kullanicilar`)
        kontrol(err)
        defer tablo.Close()
        var kullanicilar []kullanıcı
        for tablo.Next() {
            var k kullanıcı
            err := tablo.Scan(&k.id, &k.kullanici, &k.şifre,
&k.tarih)
            kontrol(err)
            kullanicilar = 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)
}
}
```

sqlite3

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 veri tabanı olaylarına geçiş yapmış olacağız.

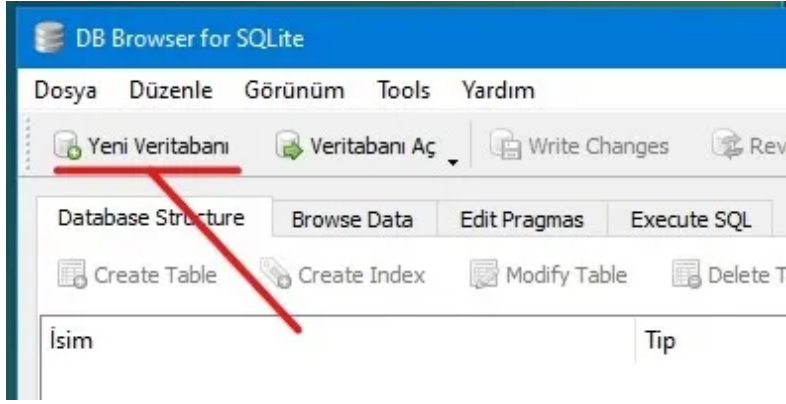
DB Browser programını aşağıdaki adresten indirebilirsiniz.
<https://sqlitebrowser.org/dl/>

Linux sistemlerin çoğunda **uygulama deposunda** bulunan bir uygulamadır.



DB Browser ikonu

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.

Tablo tanımlama penceresi (Screenshot):

Tablo: **kisiler**

Gelişmiş

Alanlar:

Alan Ekle, Alanı kaldır, Alanı yukarı taşı, Alanı aşağı taşı

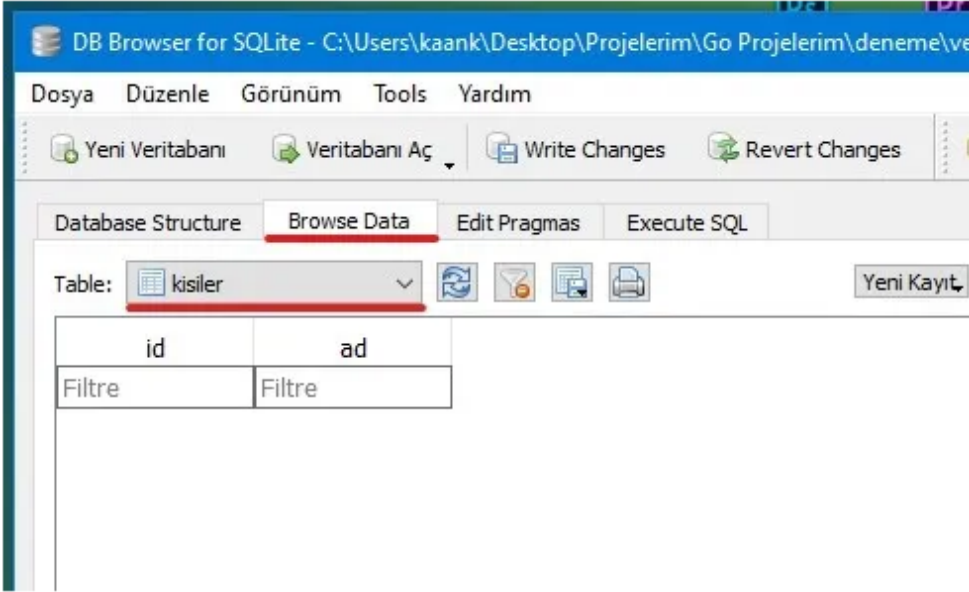
İsim	Tip	NN	Birinc	Otom	Benze	Varsayılan	Kontrol
id	INTEGER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
ad	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		

```
1 CREATE TABLE "kisiler" (  
2     "id"    INTEGER PRIMARY KEY AUTOINCREMENT,  
3     "ad"    TEXT  
4 );
```

OK Cancel

Tablomuzun ismini **kisiler** olarak ayarlayalım. **Alan Ekle**'ye tıklayarak yukarıdaki gibi **id** ve **ad** isiminde 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 veri tabanı oluşturmuş 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 kayıta 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 veri tabanı 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 numarasını 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 kontrol 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
}
```


MongoDB

MongoDB 2009 yılında geliştirilmiş açık kaynak kodlu bir NoSQL veritabanıdır. Şimdi Go ile MongoDB veritabanını kullanmak için mongo-driver paketini indirelim.

```
go get go.mongodb.org/mongo-driver/mongo
```

Kodu sürekli tekrardan yapıştırmamak için import edilecek tüm kütüphaneler burada :

```
package main

import (
    "context"
    "log"
    "time"

    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/bson/primitive"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
)
```

Şimdi de database ile bağlantıyı sağlayalım. Eğer database adresini bilmiyorsanız mongo konsolunuzun nereye bağlandığına bakarak database adresini bulabilirsiniz.

```
databaseURL := "mongodb://127.0.0.1:27017"
ctx, _ := context.WithTimeout(context.Background(),
10*time.Second)
client, err := mongo.Connect(ctx,
options.Client().ApplyURI(databaseURL))
if err != nil {
    log.Fatal("Hata : " + err.Error())
}
kisiler := client.Database("test").Collection("kisiler")
```

Bu kısım main fonksiyonunun başlangıcı. Kısaca yapılan ise ilk satırda database adresini verdik. Ardından 10 saniye timeout'u olan bir context açtık. Context ve database adresini kullanarak bağlantımızı gerçekleştirdik. Son satırda da istediğimiz veritabanı ve collection'u çektik. Bundan sonra isim adında bir struct oluşturalım. Kişinin ismini ve yaşını tutsun.

```
type Kisi struct {
    ID    primitive.ObjectID `bson:"_id,omitempty"
    json:"id,omitempty"`
    Isim  string             `bson:"isim,omitempty"
    json:"isim,omitempty"`
    Yas   int
    `bson:"yas,omitempty" json:"yas,omitempty"`
}
```

Şimdi de bir kişi oluşturup database'e ekleyelim.

```
birisi := Kisi{
    Isim: "Ahmet",
    Yas:  42,
}
res, err := kisiler.InsertOne(context.TODO(), birisi)
if err != nil {
    log.Fatal("Hata : " + err.Error())
}

id := res.InsertedID
fmt.Println(id)
```

Kişi oluşturma ve ekleme işlemlerinden sonra eklenen verinin id'sini de yazdırdık. MongoDB Compass ya da konsol üzerinden db.kisiler.find() yaparak eklenen veriyi görebiliriz. Şimdi de veri okuma yapalım. Tüm verileri almak için :

```
ctx, _ = context.WithTimeout(context.Background(),
30*time.Second)
cur, err := kisiler.Find(ctx, bson.D{})
if err != nil {
```

```

        log.Fatal("Hata : " + err.Error())
    }
    defer cur.Close(ctx)
    var list []Kisi
    for cur.Next(ctx) {
        var result Kisi
        err := cur.Decode(&result)
        if err != nil {
            log.Fatal("Hata : " + err.Error())
        }
        list = append(list, result)
    }
    if err := cur.Err(); err != nil {
        log.Fatal("Hata : " + err.Error())
    }
    fmt.Println(list)

```

Şimdi de her hangi bir özelliğe göre arama yapalım.

```

var result Kisi
ctx, _ = context.WithTimeout(context.Background(),
5*time.Second)
err = kisiler.FindOne(ctx, bson.M{"yas": 42}).Decode(&result)
if err != nil {
    log.Fatal("Hata : " + err.Error())
}
out, err := json.Marshal(&result)
if err != nil {
    log.Fatal("Hata : " + err.Error())
}
fmt.Println(string(out))

```

Silme işlemi yapalım.

```

ctx, _ = context.WithTimeout(context.Background(),
5*time.Second) // Context for Delete
_, err = kisiler.DeleteOne(ctx, bson.D{{"yas", 39}})
// Delete User
if err != nil {
    log.Fatal("Hata : " + err.Error())
}

```

İsterseniz başka bir değer üzerinden de silme işlemi yapabilirsiniz. Ben direk istenen veriyi silmek için id

kullanmanızı tavsiye ediyorum. Şimdi de Update ile temel fonksiyonları bitirelim.

```
degisecek := Kisi{
    Isim: "Kemal",
    Yas: 39,
}
yeni := Kisi{
    Isim: "Kemal",
    Yas: 100,
}
var filtre bson.M
bytes, err := bson.Marshal(degisecek)
if err != nil {
    log.Fatal("Hata : " + err.Error())
}
bson.Unmarshal(bytes, &filtre)
var usr bson.M
bytes, err = bson.Marshal(yeni)
if err != nil {
    log.Fatal("Hata : " + err.Error())
}
bson.Unmarshal(bytes, &usr)
update := bson.D{
    {"$set", usr},
}
ctx, _ = context.WithTimeout(context.Background(),
5*time.Second)
_, err = kisiler.UpdateOne(ctx, filtre, update)
if err != nil {
    log.Fatal("Hata : " + err.Error())
}
```

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 çalıştırılacağını belirliyor. **ListenAndServe()** fonksiyonu ise sunucunun ayağa kalkmasını ve istediğimiz bir porttan ulaşılmasını sağlıyor. Eğer sunucuya dosya verme yoluyla işlem yapmasını

istiyorsak aşağıdaki yöntemle 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>
```

Şimdi 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.

HTML Şablonlar (Templates)

HTML Şablonlar, Golang üzerinde web sayfalarının dinamikliği için kullanılır. Yani şablonlar kullanarak web sayfalarımızın belirlediğimiz bölümlerini Go üzerinden değişikliğe uğratabiliriz.

Bu yazımızda HTML şablonların nasıl oluşturulacağına bakacağız. Çalışma mantığı çok basit. Şablon olarak kullanacağımız html dosyasında sadece tasarımı yapıyoruz ve dinamik olacak kısımlara ise bir nevi işaretler koyuyoruz. Daha sonra Go bu şablon dosyasını işliyor ve işaret koyduğumuz yerlere gelecek değerleri yerleştiriyor. Düz mantık olarak bu işi yapıyor.

□ Örnek Kullanım

Merhaba, `{{ . }}`

Yukarıdaki örnekte `{{ }}` süslü parantezler içerisinde `.` (nokta) yazıyor. Bu da Go şablon işlenirken bu kısma Go tarafından vereceğimiz değerin geleceği anlamına geliyor.

Şimdi yukarıdaki örneğimizi `şablon.html` adı ile kaydedelim.

Gelelim Go kodlarımıza;

`main.go` dosyamız aşağıdaki gibi olsun.

```
package main
```

```
import (  
    "fmt"
```



```

    "html/template"
    "net/http"
)

// yakalayıcı fonksiyonumuz
func anasayfa(w http.ResponseWriter, r *http.Request) {
    //isim değişkenimiz
    isim := "Kaan"

    //burada şablon oluşturuyoruz
    şablon, _ := template.ParseFiles("şablon.html")

    //Burada da şablonu çalıştırmasını ve isim
    //değişkenini kullanmasını istiyoruz.
    şablon.Execute(w, isim)
}
func main() {
    fmt.Println("Sunucu Başladı")

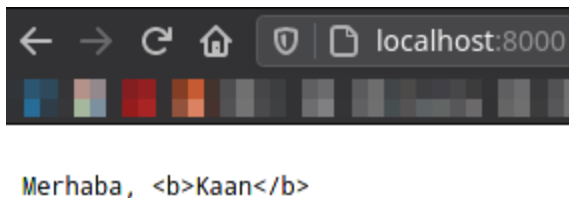
    //ana dizini anasayfa fonksiyonu ile yakalayalım
    http.HandleFunc("/", anasayfa)

    //portu 8000 yapalım ve sunucuyu başlatalım
    http.ListenAndServe(":8000", nil)
}

```

Açıklamaları üstte yazıyor.

Sayfamıza bakmak için <http://localhost:8000> adresine gittiğimizde, şöyle bir sonuç ile karşılaşacağız.



İlk Çıktımız

Tabi ki burada bir terslik var. b etiketleri gözüküyor. Bunun sebebi tarayıcımızın sayfayı html olarak değil de metin

dosyası olarak göstermesi. Çözüm için `sablon.html` dosyamızın başına `<!DOCTYPE html>` ekleyelim. Yani şöyle olacak:

```
<!DOCTYPE html>
Merhaba, <b>{{ . }}</b>
```

{% hint style="info" %} Eğer sadece şablon dosyasında değişiklik yaptıysanız, sunucuyu yeniden başlatmanıza gerek yoktur. Şablon dosyalarındaki değişiklik sunucu açıkken de güncellenir. {% endhint %}

Sayfayı yenileyerek değişikliğe bakalım. Çıktımız şöyle olacaktır:



Doğru çıktımız

Bu sefer doğru bir çıktı üretmiş olduk.

Bu yöntemler ile sayfamızda istediğimiz bölüme istediğimiz tipte değerler gönderebiliriz.

□ HTML Kodu Gönderme

Bu örneğimizde `sablon.html` dosyamız aşağıdaki gibi olsun.

```
<!DOCTYPE html>
{{ . }}
```

`main.go` dosyamız da aşağıdaki gibi olsun.

```
package main
```

```
import (
```

```

    "fmt"
    "html/template"
    "net/http"
)

func anasayfa(w http.ResponseWriter, r *http.Request) {
    //html kodumuz
    htmlKodu := "<h1>Merhaba</h1>"

    şablon, _ := template.ParseFiles("şablon.html")

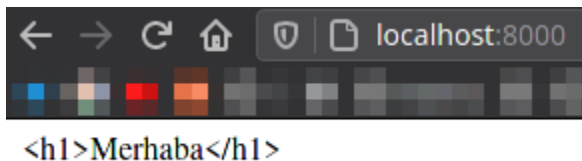
    şablon.Execute(w, htmlKodu)
}
func main() {
    fmt.Println("Sunucu Başladı")

    http.HandleFunc("/", anasayfa)

    http.ListenAndServe(":8000", nil)
}

```

Yukarıdaki örnekte sayfamıza bu sefer html kodu yolluyoruz. Çıktımıza bakalım.



HTML kodu çıktımız

Burada yine bir şeyler dönüyor. Çıktı yine istediğimiz gibi değil. “Niye bana yanlış kodları gösteriyorsun Kaan?” dediğinizi duyar gibiyim. Çünkü ilk önce yapmamamız gereken şeyleri gösteriyorum ki daha akılda kalıcı olsun.

Yukarıdaki olayın sebebi şudur: Go tarafından gönderdiğimiz html kodunun aslında `html` tipinde değil de `string` tipinde olması. Bu yüzden html kodumuz düz bir şekilde görünüyor.

Çözüm olarak da Go tarafındaki değişkenimizin tipini `template.HTML` yapacağız. `main.go` dosyamızda `htmlKodu` değişkenimizin tipini değiştirelim.

```
var htmlKodu template.HTML = "<h1>Merhaba</h1>"
```

veya burada değişkenin tipini değiştirmek yerine `şablon.Execute()` fonksiyonunda değişiklik yapabilirsiniz. (Hangisi kolayınıza geliyorsa)

```
şablon.Execute(w, template.HTML(htmlKodu))
```

□ Şablona Struct Gönderme

Buraya kadar şablon dosyamıza hep bir tane değer gönderdik. Birden fazla değer göndermek için ne yapmalıyız?

Bu konuda da struct'lerden faydalanabiliriz. Örnek olarak `main.go` dosyamız aşağıdaki gibi olsun.

```
package main

import (
    "fmt"
    "html/template"
    "net/http"
)

type bilgi struct {
    Başlık string
    İçerik template.HTML
}

func anasayfa(w http.ResponseWriter, r *http.Request) {
    sayfaBilgi := bilgi{
        Başlık: "Sayfa Başlığı",
        İçerik: "<b>sayfa içeriği</b>",
    }
    şablon, _ := template.ParseFiles("şablon.html")
    şablon.Execute(w, sayfaBilgi)
```

```
}  
func main() {  
    fmt.Println("Sunucu Başladı")  
    http.HandleFunc("/", anasayfa)  
    http.ListenAndServe(":8000", nil)  
}
```

Yukarıda `bilgi` isminde bir struct oluşturduk. Bu struct'ımız `Başlık` ve `İçerik` adında alt değişkenlere sahip. Struct'ın elemanlarını isimlendirirken baş harflerinin büyük olmasına dikkat edelim. Çünkü Şablonlar struct değişkenlerini dışa aktarma mantığı ile kullanır. Bu yüzden baş harflerini büyük yazmazsak değişkenleri sayfaya ulaştıramayız.

Daha sonra `anasayfa` fonksiyonumuz içerisinde `sayfaBilgi` adında `bilgi` struct'ı oluşturduk ve değişkenlerimizin değerlerini girdik. Son olarak `sayfaBilgi`'yi şablona gönderdik.

`sablon.html` dosyamız içerisinde ise `sayfaBilgi`'den gelen değişkenleri yerleştirelim.

```
<!DOCTYPE html>  
<h1>{{ .Başlık }}</h1><br>  
{{.İçerik}}
```

Yukarıdaki kodlarda dikkat edeceğimiz nokta, tanımlama yaparken noktadan sonra, `bilgi` struct'ının alt değişkenlerinin ismi ile çağırıyor olmamız.

Çıktımızı görelim:



Sayfa Başlığı

sayfa içeriği

Şablonda struct örneği

Buraya kadar şablon mantığını az çok anladığınıza inanıyorum. Buradan sonrasını konu çok uzun olmasın diye hızlıca anlatmaya çalışacağım. Yani buradan sonra çıktıların resimlerini göstermeyeceğim.

□ Şablon İçinde Değişken Atama

Şablon içerisinde oluşturduğumuz değişkenleri tıpkı PHP'deki gibi \$ işareti ile kullanırız.

Öncelikle `main.go` dosyamız aşağıdaki gibi olsun.

```
package main

import (
    "fmt"
    "html/template"
    "net/http"
)

func anasayfa(w http.ResponseWriter, r *http.Request) {
    şablon, _ := template.ParseFiles("şablon.html")

    //şablona değer göndermeyeceğimiz için burası nil olsun.
    şablon.Execute(w, nil)
}

func main() {
    fmt.Println("Sunucu Başladı")
    http.HandleFunc("/", anasayfa)
```

```
    http.ListenAndServe(":8000", nil)
}
```

sablon.html dosyamız da aşağıdaki gibi olsun.

```
<!DOCTYPE html>
```

```
{{ $isim := "kaan" }}
{{ $isim }}
```

Yukarıda gördüğünüz gibi değişkeni tanımlarken ve kullanırken başına \$ işareti koyduk. Çıktımızda *“kaan”* yazacaktır.

□ Şablonda If-Else Kullanımı

```
{{if .Reşit}}
    Bu kişi reşittir.
{{else}}
    Bu kişi reşit değildir.
{{end}}
```

Yukarıda dikkat etmemiz gereken şey if-else’in sonuna end eklememiz gerekiyor. Sadece if olsaydı bile end eklememiz gerekir.

Bu kodları yazdıktan sonra çıkan sonuçta boşluklar (boşluk tuşunun boşluğu gibi) oluşabilir. Bunu engellemek için ise aşağıdaki gibi yapabiliriz.

```
{{if .Reşit}}
    Bu kişi reşittir.
{{- else}}
    Bu kişi reşit değildir.
{{- end}}
```

Yani boşluk oluşan bölgeniz başına - tire ekliyoruz.

□ Şablonda Range Döngüsü Kullanımı

Range döngüsü ile web sayfamızın içerisinde bir listenin sıralanmasını sağlayabiliriz.

```
<ul>
  {{range .Liste}}
    {{.}}
  {{end}}
</ul>
```

□ Şablon İçerisinde Yorum Satırı Oluşturma

Biliyorsunuz ki, HTML kodu içerisindeki yorumlar sayfa kaynağını göstererek tıklanınca gözüküyor. Eğer yorumların gözükmesini istemezsek Şablon yorumu olarak yazabiliriz.

```
{{/* Yorumu buraya yazabilirsiniz */}}
```


Statik Kütüphanesi ile Dosyaları Uygulamaya Gömme

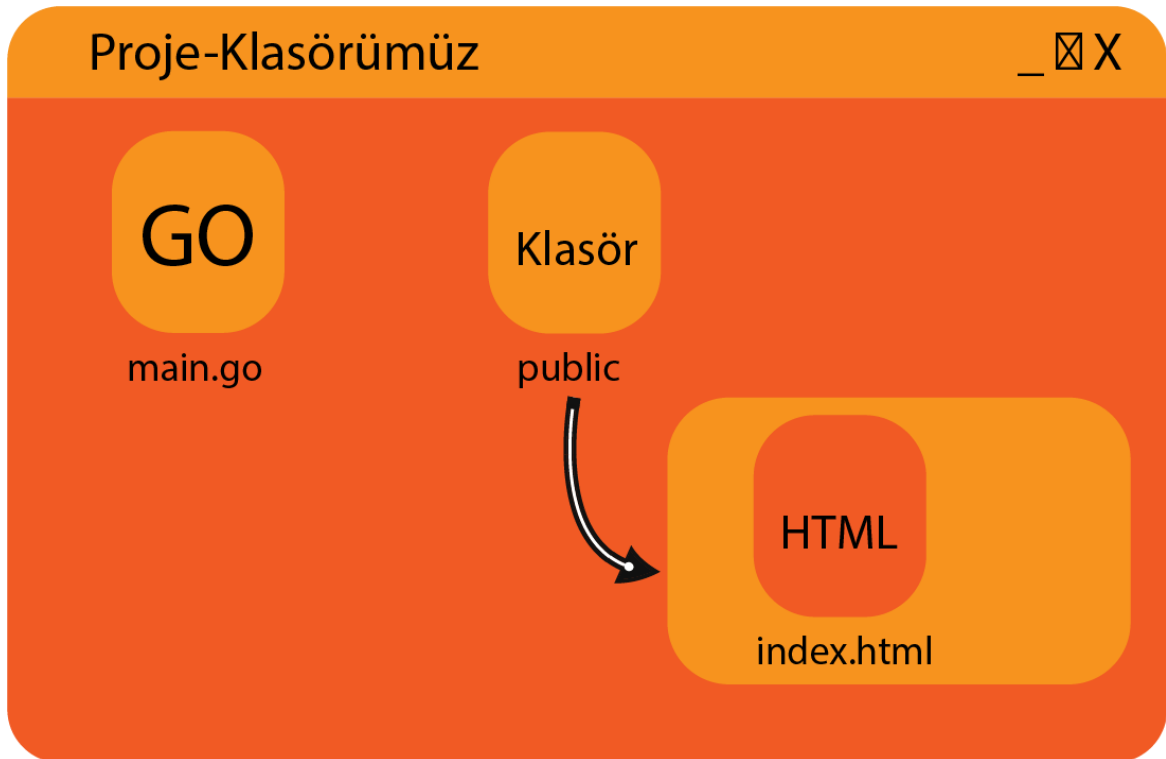
Golang'ın müthiş yanlarından biri de bir uygulamayı build ettiğimizde bize tek çalıştırılabilir dosya şeklinde sunmasıdır. Fakat bu özellik sadece **.go** dosyalarını birleştirilmesinde kullanılıyor. Harici dosyalar programa gömülmüyor. Fakat **Statik** isimli kütüphane ile bu işlem mümkün kılınıyor.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



Proje klasörümüzün dizin düzeni

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 kodlarda **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 fonksiyonel değişkendir. **_** koymamızın sebebi **error** çıktısını kullanmak istemediğimizdendir. Eğer lazım olursa kullanabilirsiniz. **fs.New()** diyerek **statikFS** değişkenimizi bir dosya sistemi olarak tanıttık. Daha sonra sunucu oluşturarak anadizine ulaşılacak istendiğinde oluşturduğumuz dosya sistemine bağlanmasını sağladık. Artık dosya sistemimize **localhost:5555** üzerinden ulaşılabilir oldu.

Gin Web Kütüphanesi

Gin Go'da yazılmış bir web kütüphanesidir. Performans ve üretkenlik odaklıdır. Sizlere basitçe web sunucu ve api oluşturmanız için kolaylık sağlar.

Kurulum için:

```
go get -u github.com/gin-gonic/gin
```

Daha sonra yine aynı yöntemle projemize dahil edebiliriz.

```
import "github.com/gin-gonic/gin"
```

Basit bir web sunucu oluşturma örneği:

```
package main

import (
    // kütüphanemizi içeri aktaralım
    "github.com/gin-gonic/gin"
)

func main() {
    //gin'in varsayılan ayarlarında bir yönlendirici oluşturalım.
    router := gin.Default()

    //anasayfayı inde fonksiyonumuz yakalayacak
    router.GET("/", index)

    //daha sonra sunucuyu başlatıyoruz
    router.Run()
}

//anasayfayı yakalayacak olan fonksiyonumuz
func index(c *gin.Context) {
    //c ile gin nesnemize bağlam oluşturduk.
    //c'yi kullanarak artık gin özelliklerine erişebiliriz.

    //sayfaya düz yazı gönderdik
    c.String(200, "Merhaba Dünya")
    //Buradaki 200 sunucudan bir cevap geldiğini anlamına gelir
}
```

Programımızı çalıştırdığımızda aşağıdaki gibi konsol çıktısı alacağız.

```
[GIN-debug] [WARNING] Creating an Engine instance with  
the Logger and Recovery middleware already attached.
```

```
[GIN-debug] [WARNING] Running in "debug" mode. Switch to  
"release" mode in production.
```

- using env: `export GIN_MODE=release`
- using code: `gin.SetMode(gin.ReleaseMode)`

```
[GIN-debug] GET / -> main.index (3 handlers)
```

```
[GIN-debug] Listening and serving HTTP on :8080
```

Bu çıktıyı incelediğimizde, Gin'in debug (hata ayıklama) modunda çalıştığını söylüyor ve hemen aşağısında sunucumuz ürün haline gelince Gin'i Release Moduna nasıl alacağımızı gösteriyor. Son olarak ise web sunucumuzun 8080 portunda çalıştığını gösteriyor.

Yukarıdaki örnekte web sunucumuz varsayılan olarak 8080 portunda çalışacaktır. Bunun sebebi `router.Run()`'a parametre olarak port numarası vermememizdir.

Örneğe göre <http://localhost:8080> adresine gittiğimizde komut satırında yeni detaylar belirecek. Tıpkı aşağıdaki gibi:

```
[GIN] 2020/11/20 - 21:34:12 | 200 | 30.253µs | ::1 | GET | "/"  
[GIN] 2020/11/20 - 21:34:12 | 404 | 1.258µs | ::1 | GET | "/favicon.ico"
```

Komut satırı bilgisi

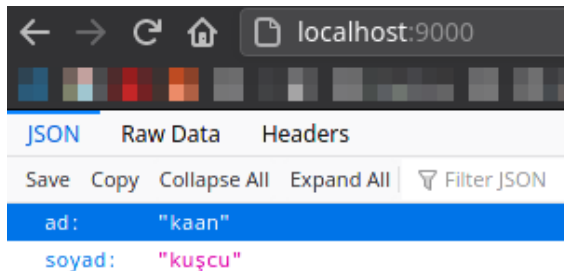
Bu bilgileri inceleyelim. İlk kayıt anasayfaya bağlanılmaya çalışıldığında alınmış. Bu kayıta bağlantının zamanını, durum kodunu, bağlantı süresi, bağlantı yöntemini ve hangi adrese bağlantı denendiğini yazıyor. Hemen altındaki ise sitenin ikonuna istek yapmış fakat site ikonumuz bulunmadığı için **404 durum kodu** almış. Bu kısımdan da bağlantı isteklerini görebildiğimizi öğrenmiş olduk.

□ Çıktı Tipleri

→ JSON Çıktı Verme

```
func index(c *gin.Context) {  
    //JSON Fonksiyonunu kullanıyoruz.  
    c.JSON(200, gin.H{  
        "ad": "kaan",  
        "soyad": "kuşcu",  
    })  
}
```

Sonucumuz aşağıdaki gibi olacaktır.



JSON çıktısı

→ XML Çıktı Verme

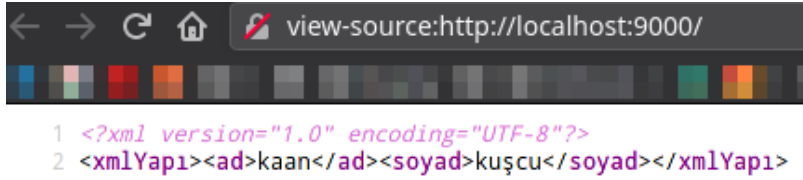
//xml için örnek bir yapı oluşturalım

```
type xmlYapı struct {  
    Ad    string `xml:"ad"`  
    Soyad string `xml:"soyad"`  
}
```

//anasayfayı yakalayacak olan fonksiyonumuz

```
func index(c *gin.Context) {  
    //xml için örnek bir nesne oluşturduk  
    xmlOrnek := xmlYapı{"kaan", "kuşcu"}  
  
    //xml başlığını gönderelim  
    c.Writer.WriteString(xml.Header) //<?xml version="1.0"  
    encoding="UTF-8"?>  
  
    //xml nesnesini XML fonksiyonu ile yolladık  
    c.XML(200, xmlOrnek)  
}
```

Bu kodlar sonucunda sayfamızı açtığımızda “kaankuşcu” sonucu göreceğiz. XML tipinde görmek için sayfanıza sağ tıklayıp “*Sayfa Kaynağını Gör*”e tıklayarak kontrol edebilirsiniz.



XML sonucu

→ Template Kullanımı

Template hakkında bilginiz yoksa önce aşağıdaki dökümanı okumanız önerilir.

[HTML Şablonlar \(Templates\)](#)

Gin’de template (şablon) işlemleri bayağı kolaylaştırılmış. Ufak bir örnek uygulama yazalım. Öncelikle projemizin ana dizinine templates isimli bir klasör oluşturalım ve içerisine index.html dosyası oluşturalım. index.html dosyamızın içeriği ise aşağıdaki gibi olsun.

```
<html>
  <h1>
    {{ .başlık }}
  </h1>
</html>
```

Burada {{ .başlık }} yerine Go’dan değer göndereceğiz.

main.go dosyamız ise aşağıdaki gibi olsun.

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
```

```

router := gin.Default()

//Burada templates klasörünün içindeki tüm şablonları
//yüklemesini isteyelim.
router.LoadHTMLGlob("templates/*")

router.GET("/", index)

router.Run()
}

func index(c *gin.Context) {

    //HTML şablonunu almak için
    //HTML fonksiyonunu kullanıyoruz.
    c.HTML(200, "index.html", gin.H{
        //Şablondaki başlık yerine Anasayfa yazısını yollayalım.
        "başlık": "Anasayfa",
    })
}

```

Web sunucumuza bağlandığımızda ise **Anasayfa** yazdığını görebiliriz.

□ Statik Dosyaları Yayınlama

Web sunucumuzda kullanacağımız Css, JS vb. statik dosyalarımız olabilir. Bunun için Static fonksiyonunu kullanabiliriz.

Statik dosyalarımızı projemizin ana dizindeki statik klasöründe barındırdığımızı varsayalım.

```

func main() {
    router := gin.Default()

    //(yönlendirme, klasör-ismi)
    router.Static("/static", "./statik")

    router.GET("/", index)

    router.Run(":9000")
}

```

statik klasörümüzün içerisinde index.js adında bir dosya olduğunu varsayarsak <http://localhost:9000/static/index.js>

adresinden ulaşabiliriz.

□ Bağlantı Metodları

Örnek bağlantı oluştururken GET metoduna değindik. Metodları test ediyorken **Postman**'i kullanabilirsiniz. Ben bu konuda **curl** komut satırı aracını kullanacağım. Detaylarına bakacak olursak:

➔ GET Metodu

GET metodu web sunucumuza normal bağlantı yapılırken kullanılır. Hazır bir kaynağı yüklemek için kullanılır.

```
router.GET("/", index)
```

index fonksiyonu ile GET metodlu anasayfayı yakalayabilirsiniz.

➔ POST Metodu

POST metodu genellikle form gönderimlerinde kullanılır. Yeni bir kaynak oluşturmak için kullanılır. (Yeni kayıt oluşturma, yeni gönderi oluşturma vb...) Örnek kullanımını görelim.

```
package main
```

```
import (  
    "github.com/gin-gonic/gin"  
)
```

```
func main() {  
    router := gin.Default()  
  
    //Burada templates klasörünün içindeki tüm şablonları  
    //yüklemesini isteyelim.  
    router.LoadHTMLGlob("templates/*")  
  
    router.GET("/", getIndex)  
    router.POST("/", postIndex)  
  
    router.Run(":9000")  
}
```

```
func getIndex(c *gin.Context) {
```

```

    c.String(200, "GET metodu ile bağlanıldı.")
}

func postIndex(c *gin.Context) {
    c.String(200, "POST metodu ile bağlanıldı.")
}

```

Yukarıdaki örnekte anasayfa GET ile bağlanıldığında `getIndex`, POST ile bağlanıldığında `postIndex` fonksiyonu çalışacak. Tarayıcımızdan girdiğimizde “*GET metodu ile bağlanıldı.*” yazısını görürüz. POST metodu ile bağlanmak için komut satırına şu komutları yazalım.

```
curl -X POST http://localhost:9000
```

Çıktısı “*POST metodu ile bağlanıldı.*” olacaktır.

POST metodu üzerinden değer almayı görelim.

```

//json verisi için yapımız
type kişi struct {
    Ad      string `json:"ad"`
    Soyad   string `json:"soyad"`
}

func postIndex(c *gin.Context) {
    //posttan gelen json'ın kaydedileceği değişken
    var postkişi kişi

    //postan gelen json'ı postkişi'ye atayalım
    c.BindJSON(&postkişi)

    c.String(200, "JSON Veri:")
    //json'ı tekrar post ile gösterelim
    c.JSON(200, postkişi)
}

```

Komut satırına aşağıdaki komutu yazarak çıktısını görebilirsiniz.

```
curl -X POST -H "Content-Type: application/json" -d
'{"ad":"kaan","soyad":"kuşcu"}' http://localhost:9000
```

➔ Diğer Metodlar

Diğer metodlardan kısaca bahsedelim:

- **PATCH metodu:** Bir kaynak üzerindeki belirli bir alanı değiştirmek için kullanılır.
- **DELETE metodu:** Sunucudaki bir kaynağı silmeye yarar.
- **PUT metodu:** Bir kaynağın yerine başka bir kaynağı koymaya yarar. (Komple değiştirme)
- **HEAD metodu:** Sunucuya tıpkı GET metodu gibi fakat sadece başlığı olan bir istek gönderir.
- **OPTIONS metodu:** Sunucunun desteklediği metodları kontrol etmek için kullanılır.

□ Adreslendirme

➔ Parametre ile Adreslendirme

Örneğin:

`http://localhost:9000/blog/yazı-ismi`

değişken kısım

Gin parametre örneği

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
    router := gin.Default()

    router.GET("/blog/:yazı", blog)
    //Buradaki :yazı bizim parametremiz
    //Bu parametre ile hangi blog yazısını
    //göstereceğimizi belirleyeceğiz.

    router.Run(":9000")
}

func blog(c *gin.Context) {
```

```
//yazı parametresinde geleni sayfaİsmi değişkenine atayalım.  
sayfaİsmi := c.Param("yazı")  
c.String(200, "Şuanda "+sayfaİsmi+" blogunu okuyorsunuz.")  
}
```

Yukarıdaki örneğe göre

<http://localhost:9000/blog/gin%20ile%20sunucu%20geli%C5%9Ftirme> adresine gittiğimizde “*Şuanda gin ile sunucu geliştirme blogunu okuyorsunuz.*” yazısı ile karşılaacağız.

Tabi ki birden fazla parametre ekleyebilirsiniz.

```
package main  
  
import (  
    "github.com/gin-gonic/gin"  
)  
  
func main() {  
    router := gin.Default()  
  
    router.GET("/blog/:yazar/:yazı", blog)  
  
    router.Run(":9000")  
}  
func blog(c *gin.Context) {  
    yazar := c.Param("yazar")  
    yazı := c.Param("yazı")  
    c.String(200, "Yazar: "+yazar+" Yazı: "+yazı)  
}
```

→ Querystring (Sorgu dizesi) ile Adreslendirme

Örneğin:

<http://localhost:9000/arama> **?tur=bilim-kurgu&siralama=imdb**



değişken kısım

Gin sorgu sizesi örneği

```
package main  
  
import (  

```

```
    "github.com/gin-gonic/gin"
)

func main() {
    router := gin.Default()

    router.GET("/arama", arama)

    router.Run(":9000")
}

func arama(c *gin.Context) {
    tür := c.Query("tur")
    sıralama := c.Query("siralama")
    c.String(200, tür+" türünden filmler "+sıralama+" olarak
sıralanıyor.")
}
```

Yukarıdaki örneğe göre <http://localhost:9000/arama?tur=bilim-kurgu&siralama=imdb> adresine girdiğimizde “*bilim-kurgu türünden filmler imdb olarak sıralanıyor.*” yazılı bir sonuç elde edeceğiz.

Göz atmanızı önerdiğim yazı:

[Go Web Sunucuda Kullanıcı Girişi Sistemi](#)

Gin Dosya Yükleme

Bu yazıda Gin kütüphanesinden POST isteği ile nasıl dosya yükleyeceğimizi göreceğiz.

Aşağıda projemizin dizin/dosya yapısı bulunuyor.

.

└─ main.go

└─ public

└─ index.html

İlk olarak index.html dosyamızı görelim.

```
<!doctype html>
```

```
<html lang="tr">
```

```
<head>
```

```
<meta charset="utf-8">
```

<title>Dosya Yükle</title>

</head>

<body>

<h1>Yüklemek için 1 adet dosya seçin</h1>

<form action="/yukle" method="post"
enctype="multipart/form-data">

İsim: <input type="text" name="isim">

E-posta: <input type="email" name="email">

Dosya: <input type="file" name="dosya">

<input type="submit" value="Yükle">

</form>

</body>

Detaylıca değinmiyorum. input elementlerinin name özelliğinde ne isim verdiğimizizi dikkat etmemiz yeterli. form elementinin action özelliğine /yukle adresini verdik.

main.go dosyamız ise aşağıdaki gibi olacaktır. Detaylarına yorum satırlarında değindim.

```
package main

import (
    "fmt"
    "path/filepath"

    "github.com/gin-gonic/gin"
)

func main() {

    router := gin.Default()
    //Yükleme yapacağımız dosyanın maksimum boyutunu
    //ayarlayalım.
    router.MaxMultipartMemory = 8 << 20 // 8 MiB

    //index.html dosyamızın bulunduğu klasörü static olarak
    //ekliyoruz ki index.html dosyamızı kullanabilelim.
    router.Static("/", "./website")

    //yukleme işlemini yapacak olduğumuz yönlendirmeyi
    //buradan ayarlıyoruz.
    router.POST("/yukle", func(c *gin.Context) {

        //form içerisindeki dosya hariç verilerimizi
        //atayalım
        isim := c.PostForm("isim")
        eposta := c.PostForm("email")

        // Dosyamızı ise özel fonksiyon ile atıyoruz
        dosya, hata := c.FormFile("dosya")

        //Hata kontrolü yapmayı unutmayalım.
        if hata != nil {
            c.String(400, fmt.Sprintf("Form Hatası: %s",
```



```

hata.Error()))
    return
}

//yüklenecek olan dosyanın ismini alalım.
dosyaismi := filepath.Base(dosya.Filename)

//yüklenen dosyanın kaydedileceği konum
// dizin + dosyaismi
kayıtYeri := "./website/" + dosyaismi

//dosyayı kaydedelim ve hata kontrolü yapalım.
if hata := c.SaveUploadedFile(dosya, kayıtYeri); hata
!= nil {
    c.String(400, fmt.Sprintf("Dosya Yükleme Hatası:
%s", hata.Error()))
    return
}

//Şuana kadar herhangi bir sıkıntı ile
//karşılaşmadıysak, olumlu mesajımızı gösterelim.
c.String(200, fmt.Sprintf("%s isimli dosya başarıyla
yüklendi \nİsim: %s\nE-posta: %s", dosya.Filename, isim,
eposta))
})
router.Run(":8080")
}

```

gRPC

gRPC Nedir?

gRPC, Google tarafından geliştirilen, açık kaynaklı uzaktan prosedür çağırma (rpc) kütüphanesi (sistemi)'dir. Aktarım için HTTP/2 kullanır. Arayüz tanımlama dili olarak protokol tamponları (buffers) kullanır. Kimlik doğrulama, Çift yönlü akış, engelleme ve engelleme olmayan bağlantılar, iptal ve zaman aşımı işlemleri için kullanılır.

JSON ve HTTP API'lerinin aksine, daha katı bir spesifikasyona sahiptirler. Tek bir spesifikasyona sahip olduğu için, gRPC tartışmalarını ortadan kaldırır ve geliştiriciye zaman kazandırır. Çünkü gRPC, platformlar ve uygulamalar arasında tutarlıdır.

Özet olarak, uzaktan prosedür çağırarak client (müşteri) ve server (sunucu) programları arasındaki iletişimi sağlayan bir kütüphanedir.

Protokol Tamponları (Buffers) Nedir?

İki uygulama arasındaki iletişimin nasıl olacağını belirleyen sözleşmedir.

Örnek Uygulama Yapalım

Bu örneğimizde sunucuya mesaj gönderip, karşılığında mesaj alan bir uygulama yazacağız. Bir nevi chat uygulaması olacak.

Öncelikle, iletişim protokolünü Go koduna dönüştürebilmemiz (generating) için protoc'yi bilgisayarımıza kuralım.


İndirmek için [buradaki](#) adrese gidin. Buradaki versiyon v3.13.0 versiyonu. Daha güncel versiyonlar için [buradaki](#) sayfayı kontrol etmeyi unutmayın.

 [protoc-3.13.0-linux-aarch_64.zip](#)

 [protoc-3.13.0-linux-ppcle_64.zip](#)

 [protoc-3.13.0-linux-s390x.zip](#)

 [protoc-3.13.0-linux-x86_32.zip](#)

 [protoc-3.13.0-linux-x86_64.zip](#)

 [protoc-3.13.0-osx-x86_64.zip](#)

 [protoc-3.13.0-win32.zip](#)

 [protoc-3.13.0-win64.zip](#)

GNU/Linux İS kullanan arkadaşlarımız uygulama depolarından protobuf adıyla yükleyebilirler. Böylece PATH olarak eklemelerine gerek kalmaz.

Sisteme Yol Olarak Ekleme

Eğer GitHub sayfasından indirdiyseniz, sisteminize dosya yolunu tanıtmanız gerekir. Bunun için örnek olarak;

Windows üzerinde

Örnek olarak protoc.exe dosyamız C:\\protoc\\bin klasörü içerisinde olsun.

Başlar menüsünde “*Ortam değişkenleri*” yazarak aratalım. Açılan pencerede *Gelişmiş* sekmesinde alt tarafta *Ortam Değişkenleri* butonuna basalım.

PATH seçeneğini *Düzenle* diyerek protoc.exe’nin konumunu ekleyelim.

GNU/Linux, MacOS vs. Bash Kullanan Sistemler Üzerinde

protoc çalıştırılabilir dosyamızı örnek olarak Home (Ev) dizinine atmış olalım.

Örnek: ~/protoc/bin olsun.

Ev dizinimizdeki .bashrc dosyamızın en altına şunları ekleyelim.

```
export PATH="~/protoc/bin:$PATH"
```

.bashrc dosyasını kaydettikten sonra komut satırına source ~/.bashrc yazarak dosyada yaptığımız değişimi onaylayalım.

Protoc'yi kontrol edelim

Komut satırına aşağıdakileri yazarak protoc'nin versiyonuna bakalım.

```
protoc --version
```

Protoc'ye Golang desteğini eklemek için aşağıdaki paketleri kuralım.

```
go get google.golang.org/protobuf/cmd/protoc-gen-go
go get google.golang.org/protobuf/cmd/protoc-gen-go-grpc
```

Aynı şekilde Windows üzerinde C:\\Users\\isim\\go\\bin klasörünü ekli değilse ortam değişkenlerine ekleyelim. Unix-like sistemler için ~/go/bin'i path'e ekleyelim.

Yukarıda belirttiğimiz örnek chat uygulamasını yazmak için aşağıdaki gibi bir proje yapımız olacak.

```
.
├── chat
│   └── chat.go
├── chat.proto
├── client.go
└── server.go
```

Client ve Server arasındaki iletişim protokolünü belirleyelim.

chat.proto dosyamız şöyle olsun.

```
//Söz dizimi için proto3'ü kullanacağız
syntax = "proto3";
```

```
//Go paketi için ayarlarımız
package chat;
option go_package=".;chat";

//iletişimde kullanılacak mesajın yapısı
message Message {
    string body = 1;
    //buradaki 1 sıra numarasıdır. Değer ataması değildir.
}

//rpc servisimiz
service ChatService {
    rpc SayHello(Message) returns (Message) {}
}
```

Yukarıdaki chat.proto dosyasını Go koduna dönüştürelim. Bunun için proje dizinindeyken aşağıdaki komutu yazalım.

```
protoc -go_out=plugins=grpc:chat chat.proto
```

Bu işlem sonucunda chat klasörümüzün içerisinde chat.pb.go dosyamız oluşacak. Bu dosyanın en başında yorum olarak düzenleme yapmamızın uygun olamayacağı yazıyor. O yüzden bu dosyayla çok uğraşmamakta fayda var.

Server (Sunucu) Oluşturalım

chat klasörümüzdeki chat.go dosyasını oluşturalım.

```
package chat

import (
    "log"

    "golang.org/x/net/context"
)

type Server struct {
    //Burası şuanlık boş olacak
}

//server nesnemize iliştirilecek fonksiyonumuz
func (s *Server) SayHello(ctx context.Context, message *Message)
(*Message, error) {
```

```
//Client'ten gelen mesajı ekrana bastıralım
log.Printf("Clientten mesaj alındı: %s", message.Body)

//mesaj gelince değer döndürelim (mesaj ve hata)
return &Message{Body: "Server'dan merhaba!"}, nil //hata nil
olsun
}
```

Go modules'tan faydalanarak chat paketini diğer paketlerde kullanabiliriz. Bunun için proje dizinindeyken komut satırına:

```
go mod init github.com/ksckaen1/grpcOrnek
```

yazalım. Buradaki yaptığımız şey, GitHub'a paket yüklemeyen sanal olarak yazdığımız chat paketini kullanabileceğiz. Bu komuttan sonra proje dizinimizde go.mod dosyası oluşacak. Böylece chat klasörünü paket olarak diğer yerlerde de kullanabileceğiz.

server.go dosyamızı oluşturalım.

```
package main

import (
    "log"
    "net"

    "github.com/ksckaen1/grpcExample/chat"
    "google.golang.org/grpc"
)

func main() {

    //tcp dinleyelim
    lis, err := net.Listen("tcp", ":9080")

    //hata kontrolü yapalım
    if err != nil {
        log.Fatalf("9080 portu dinlenirken hata oluştu: %v \n", err)
    }

    //chat server nesnesi oluşturalım
    s := chat.Server{}

    //grpc server'ı oluşturalım.
    grpcServer := grpc.NewServer()
```

```
//tcp sunucu ile grpc'yi bağlayalım.  
chat.RegisterChatServiceServer(grpcServer, &s)  
  
//hata kontrolü  
if err := grpcServer.Serve(lis); err != nil {  
    log.Fatalf("grpcServer oluşturulurken hata : %v", err)  
}  
}
```

go run server.go komutunu yazarak test edelim. Eğer çıktı vermiyorsa ve program kapanmıyorsa server dinleniyor demektir. Yani şuana kadar başarılıyız.

Client (Müşteri) Oluşturalım

Şimdi de Server'a mesaj yollayabilmek için Client'i oluşturalım. client.go dosyamız aşağıdaki gibi olsun.

```
package main  
  
import (  
    "log"  
  
    //chat klasörünü kullanacağımız için  
    //go mod init üzerinde belirlediğimiz deponun  
    //sonuna /chat ekliyoruz.  
    "github.com/ksckaanel/grpcExample/chat"  
    "golang.org/x/net/context"  
  
    //grpc kütüphanemiz  
    "google.golang.org/grpc"  
)  
  
func main() {  
  
    //grpc client bağlantı nesnesi  
    var conn *grpc.ClientConn  
  
    //grpc ye 9080 portunu dinlemesini söyleyelim  
    conn, err := grpc.Dial(":9080", grpc.WithInsecure())  
    if err != nil {  
        log.Fatalf("9080 portu dinlenirken hata oluştu: %v", err)  
    }  
  
    //bağlantıyı kapatmayı unutmayalım
```

```

defer conn.Close()

//Client nesnesi oluşturalım
c := chat.NewChatServiceClient(conn)

//server'a gönderilecek mesajı belirleyelim
message := chat.Message{
    Body: "Client'ten merhabalar!",
}

//Server'dan dönen cevabı alalım
response, err := c.SayHello(context.Background(), &message)

//hata kontrolü yapalım.
if err != nil {
    log.Fatalf("SayHello fonksiyonu çağırılırken hata oluştu:
%v", err)
}

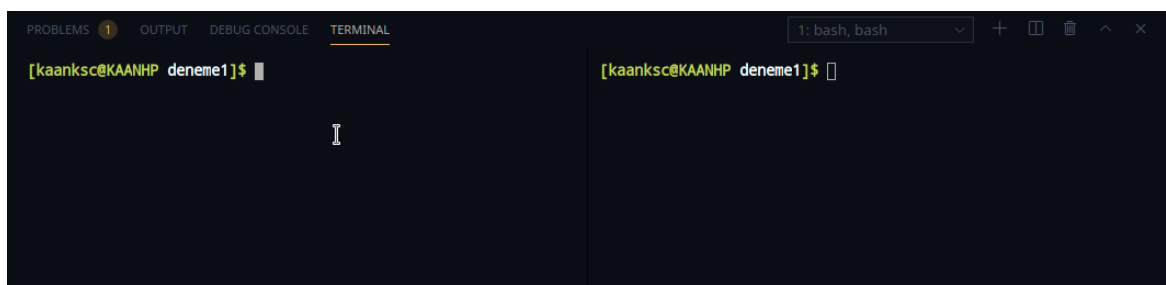
//server'dan gelen mesajı ekrana bastıralım
log.Printf("Server'dan gelen cevap: %s", response.Body)
}

```

Yukarıda yazdığımız kodların çalışma mantığını açıklayacak olursak:

1. Server 9080 portunu belirlediğimiz protokol sözleşmesine (*chat.proto*) göre dinliyor.
2. Client.go dosyamız çalıştırılınca server'a mesaj yolluyor.
3. Client'ten gelen mesajı server ekrana bastırıyor ve bunun karşılığında server client'e cevap veriyor.
4. Server'dan gelen cevabı da client ekrana bastırıyor.
5. Son olarak client programımız sonlanıyor (kapanıyor).

Gif olarak sonucu göstermek gerekirse:



gRPC Örneği

Heroku'da Go Uygulaması Yayınlama

Öncelikle Bilmeyenler İçin Heroku Nedir?

Kısaca Heroku, JavaScript, Go, Ruby, Python, Scala, PHP, Java, Clojure ile geliştirdiğimiz sunucu uygulamalarını ücretsiz barındırabileceğimiz bir platformdur.

Aşağıdaki bağlantıdaki blog yazısını okumanızı tavsiye ederim.

[Heroku Nedir? - ceaksan](#)

Projemizi Planlayalım

Bu örneğimizde bir web sunucu oluşturacağız. Öncelikle Go modules kullanacağımız için projemizi kullanıcının go dizinine oluşturmamız.

Komut satırını açalım ve aşağıdaki komutu yazarak bahsettiğimiz dizine geçelim.

Windows'ta:

```
cd C:\Users%username%
```

GNU/Linux ve MacOS'te:

```
cd ~/go/src
```

Bu konuma proje dizinimizi oluşturalım

```
mkdir heroku-app
```

heroku-app klasörü projemizin ana dizini olacak. Aşağıdaki komut ile proje ana dizinimize girelim.

```
cd heroku-app
```

Daha sonra bu dizini `code` . komutu ile VSCode üzerinde açalım.

`main.go` dosyamızı oluşturalım ve aşağıdaki gibi olsun.

```
package main

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

func main() {
    port := os.Getenv("PORT")
    http.HandleFunc("/", anaSayfa)
    http.ListenAndServe(":"+port, nil)
}

func anaSayfa(w http.ResponseWriter, r *http.Request) {
    port := os.Getenv("PORT")
    fmt.Fprintf(w, "Merhaba Dünya!\nKullanılan Port: "+port)
}
```

Yukarıda normal web sunucu oluşturma kodlarından biraz farklı işlemler var. Bunları açıklamak gerekir ise:

`port` değişkenimiz sistemden `string` tipinde `PORT` ortam değişkenini alıyor. Yani `port` değişkeni sunucumuzun hangi portta çalışacağını belirliyor. Uygulamamızı Heroku'ya yükledikten sonra sistemimiz Heroku olduğu için `port` ortam değişkenini Heroku'dan almış olacağız. Sunucunun çalışacağı portu Heroku belirlediği için portu kendimiz kodlar içinde belirleyemiyoruz.

`http.ListenAndServe()` fonksiyonuna da parametre olarak `:"+port` vererek devam ediyoruz.

Sunucumuzun ana dizinini yakalacak olan `anaSayfa` fonksiyonumuza bakalım.

Yine burada sistemden portu istedik. Hemen aşağısında "Merhaba Dünya!" ve kullanılan portun çıktısını vermesini

sağladık. Kodlarımız artık hazır.

Bu projemizde dışarıdan bir pakete ihtiyacımız olmadı. Hepsi Go'nun hazır paketlerinden. Eğer dışarıdan paketler olsa ne yapacaktık? Hadi hemen görelim.

Komut satırına go modules için aşağıdaki komutu yazalım.

```
go mod init
```

{% hint style="warning" %} Eğer projenizi go/src klasörü içinde oluşturmazsanız bu komut hata verecektir. {% endhint %}

```
[kaanksc@KAANHP heroku-app]$ go mod init  
go: creating new go.mod: module heroku-app
```

go modules örnek

Böylece go.mod dosyamızı oluşturduk. Dışarıdan paket bağımlılıklarını yüklemek için aşağıdaki komutu yazalım.

```
go mod vendor
```

Bu komutu yazdığınızda paket bağımlılığınız yoksa aşağıdaki gibi bir çıktı alacaksınız.

```
[kaanksc@KAANHP heroku-app]$ go mod vendor  
go: no dependencies to vendor
```

vendor örneği

Eğer paket bağımlılığınız varsa projenizin ana dizininde vendor adında bir klasör oluşacak ve bu klasörün içinde dış paketlerin kaynak kodları bulunacaktır.

Versiyon sistemli hale getirelim

Heroku platformu versiyon kontrol sistemi ile çalıştığı için, öncelikle git projemizi oluşturalım. Projemizin ana dizinindeyken komut satırına:

```
git init
```

Daha sonra oluşturduğumuz projeyi staging'e almak için:

```
git add .
```

yazalım. Commit oluşturmak için ise:

```
git commit -m "Heroku uygulamamı oluşturdum."
```

Heroku'da Yayınlama

Öncelikle Heroku'nun komut satırı uygulamasını bilgisayarımıza kuralım.

Windows, MacOS ve Ubuntu için [bu adresten](#) kurabilirsiniz.

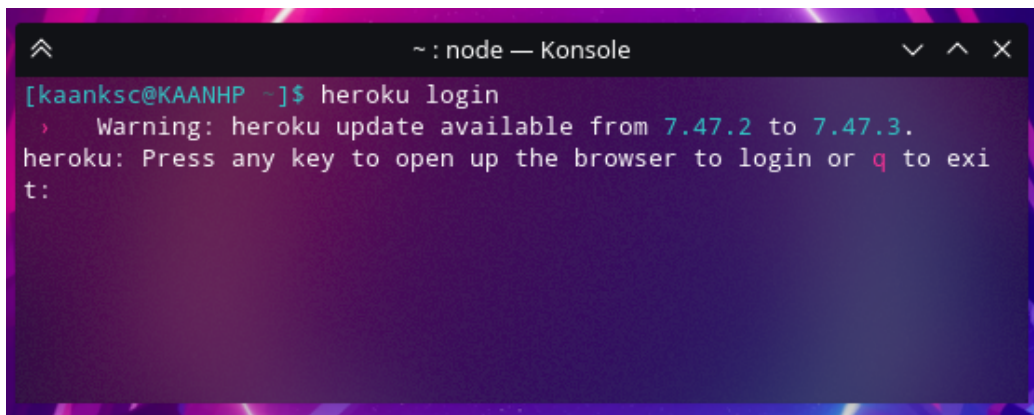
Arch Linux ve türevleri için kolaylık açısından AUR üzerinden heroku-cli-bin aratarak kurabilirsiniz.

Uygulamayı kurduktan sonra Heroku Hesabımıza bağlayalım.

Komut satırına aşağıdakileri yazalım.

```
heroku login
```

Şöyle bir çıktı verecek:



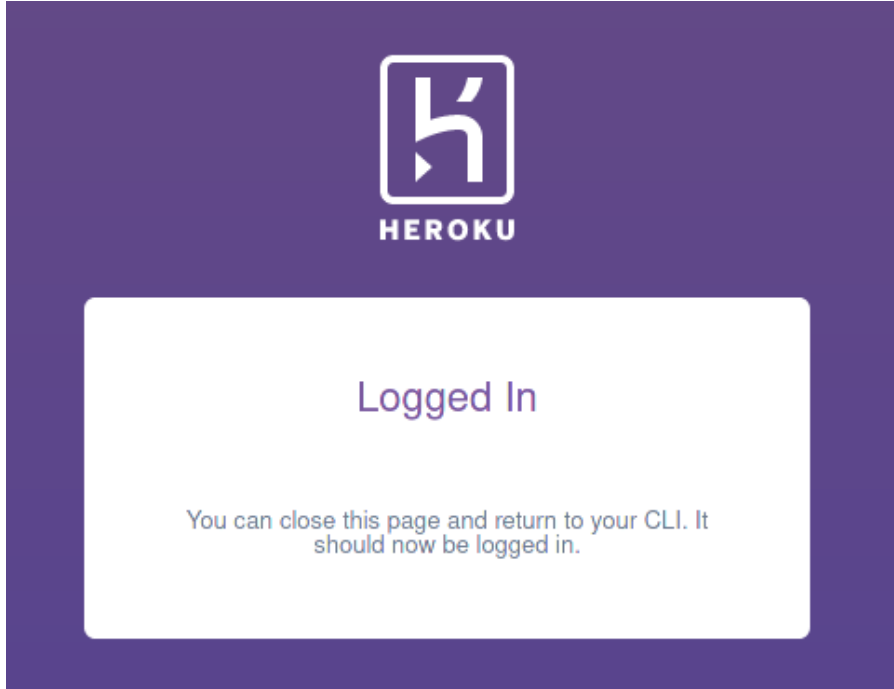
```
~ : node — Konsole
[kaanksc@KAANHP ~]$ heroku login
> Warning: heroku update available from 7.47.2 to 7.47.3.
heroku: Press any key to open up the browser to login or q to exit:
```

Heroku cli Login

q tuşuna basınca giriş yapmayı iptal eder. O yüzden giriş yapmak için herhangi bir tuşa basabilirsiniz. (Lütfen bilgisayarınızın güç

tuşuna basmayın ☐)

Daha sonra varsayılan tarayıcınız üzerinden giriş yapma sayfası açılacak. Heroku hesabınıza girdikdek sonra tarayıcınızda girişin başarılı olduğunu söylecek.



Heroku tarayıcı girişi

Komut satırında da aşağıdaki gibi bir çıktı göreceksiniz. Kendi bilgilerim olduğu için birazını sansürledim.

```
~ : bash — Konsole
> Warning: heroku update available from 7.47.2 to 7.47.3.
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/9
[redacted]
Logging in... done
Logged in as [redacted]@hotmail.com
[kaanksc@KAANHP ~]$
```

Heroku cli başarılı giriş

Böylece başarıyla giriş yapmış olduk.

Heroku projemizi oluşturalım.

heroku create

Şöyle bir çıktı alacağız.

```
[kaanksc@KAANHP heroku-app]$ heroku create
> Warning: heroku update available from 7.47.2 to 7.47.3.
Creating app... done, ● obscure-ocean-33068
https://obscure-ocean-33068.herokuapp.com/ | https://git.heroku.com/obscure-ocean-33068.git
```

heroku uygulama oluşturma

Yazdığımız kodları Heroku uygulamamıza yükleyelim.

git push heroku master

Bu komut sonrasında aşağıdakine benzer bir sonuç almanız gerekir.

```
[kaanksc@KAANHP heroku-app]$ git push heroku master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 477 bytes | 477.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Go app detected
remote: -----> Fetching jq... done
remote: -----> Fetching stdlib.sh.v8... done
remote: ----->
remote: Detected go modules via go.mod
remote: ----->
remote: Detected Module Name: heroku-app
remote: ----->
remote: !! The go.mod file for this project does not specify a
Go version
remote: !!
remote: !! Defaulting to go1.12.17
remote: !!
remote: !! For more details see:
https://devcenter.heroku.com/articles/go-apps-with-modules#build-
configuration
remote: !!
remote: -----> New Go Version, clearing old cache
```

```
remote: -----> Installing go1.12.17
remote: -----> Fetching go1.12.17.linux-amd64.tar.gz... done
remote: -----> Determining packages to install
remote:
remote:      Detected the following main packages to install:
remote:      heroku-app
remote:
remote: -----> Running: go install -v -tags heroku heroku-app
remote: heroku-app
remote:
remote:      Installed the following binaries:
remote:      ./bin/heroku-app
remote:
remote:      Created a Procfile with the following entries:
remote:      web: bin/heroku-app
remote:
remote:      If these entries look incomplete or incorrect please
remote:      create a Procfile with the required entries.
remote:      See https://devcenter.heroku.com/articles/procfile
remote:      for more details about Procfiles
remote:
remote: -----> Discovering process types
remote:      Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:      Done: 3.6M
remote: -----> Launching...
remote:      Released v3
remote:      https://obscure-ocean-33068.herokuapp.com/ deployed
remote:      to Heroku
remote:
remote: Verifying deploy... done.
remote: To https://git.heroku.com/obscure-ocean-33068.git
remote: * [new branch]      master -> master
```

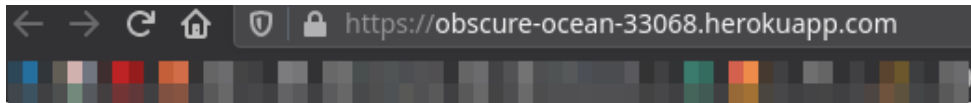
Yukarıdaki çıktıya göre aşağıdaki işaretlediğim yerde uygulamamızın adresi olacak.

```
remote: If these entries look incomplete or incorrect please create a Procfile with th
remote: See https://devcenter.heroku.com/articles/procfile for more details about Proc
remote:
remote: -----> Discovering process types
remote: Procfile declares types -> web
remote:
remote: -----> Compressing...
remote: Done: 3.6M
remote: -----> Launching...
remote: Released v3
remote: https://obscure-ocean-33068.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
to https://git.heroku.com/obscure-ocean-33068.git
```

Burada

Heroku Push sonuç

Bu adres tabiki de sizde farklı olacak. Buradan girip uygulamanızı kontrol edebilirsiniz. Benim sonucum ise şu şekilde:



Merhaba Dünya!
Kullanılan Port: 38383

Site sonucu

HTTP İstekleri (Requests)

Bu bölümde bir web sayfasına nasıl istekte bulunacağımızı göreceğiz.

Get İsteği

Standart istektir.

```
package main

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

func main() {
    //Google anasayfasına GET isteğinde bulunalım
    //ve dönen cevabı cevap değişkenine atayalım
    cevap, hata := http.Get("https://www.google.com.tr")
    if hata != nil {
        //Ya sev ya terket dedik ve paniği bastık.
        panic(hata)
    }
    //Body'i kapatmayı unutmayalım
    defer cevap.Body.Close()

    //Bu esnada Body'den cevabı çekip sayfa değişkenine
    alıyoruz.
    //Çünkü bunu yapmadan okunabilir bir sonuç alamayız.
    sayfa, hata := io.ReadAll(cevap.Body)

    if hata != nil {
        panic(hata)
    }

    //sayfa değişkeni byte dizisi tipinde olduğu için
    //string tipine çevirip okuyalım
    fmt.Println(string(sayfa))
}
```

```
    //Çıktımız Google TR anasayfasının kaynak kodları olacaktır.  
}
```

Sorgu Parametresi Ekleme

Örneğin, yaptığımız istekte `search?q=golang&hl=de` gibi bir sorgu kısmı olmasını istiyoruz. URL'yi girdiğimiz kısmı bunu el ile ekleyebiliriz tabiki. Ama onunda kolay bir yöntemi var. Bunun için `net/url` paketini içe aktaralım. Hemen görelim.

```
func main() {  
    params := url.Values{  
        "q": {"golang"}, //aranacak metin  
        "hl": {"tr"}, //örnek olarak Türkçe sonuç vermesi için  
    }  
    fmt.Println("https://www.google.com.tr/search?" +  
params.Encode())  
}
```

Çıktımız: `https://www.google.com.tr/search?q=golang&hl=tr` olacaktır.

Bu örneği de aşağıdaki gibi kullanabilirsiniz.

```
package main  
import (  
    "fmt"  
    "io"  
    "net/http"  
    "net/url"  
)  
  
func main() {  
    params := url.Values{  
        "q": {"golang"},  
        "hl": {"tr"},  
    }  
  
    //URL'imizi aşağıdaki gibi yazalım.  
    cevap, hata := http.Get("https://www.google.com.tr/search?"  
+ params.Encode())  
    if hata != nil {
```

```

        panic(hata)
    }
    defer cevap.Body.Close()

    sayfa, hata := io.ReadAll(cevap.Body)

    if hata != nil {
        panic(hata)
    }

    fmt.Println(string(sayfa))
}

```

Postform İsteği

```

package main

import (
    "fmt"
    "io"
    "net/http"
    "net/url"
)

func main() {
    //URL'e gönderilecek verileri oluşturalım.
    değerler := url.Values{
        "isim": {"Kaan"},
        "soyisim": {"Kuşcu"},
    }

    //değerler.Encode() yapmadan direkt değerler şekline
    gönderiyoruz
    //çünkü göndereceğimiz bir url parametresi değildir.
    cevap, hata :=
    http.PostForm("https://orneksite.com/kisiler", değerler)

    //Ya sev ya terket
    if hata != nil {
        panic(hata)
    }
    //Body'i kapatmayı unutmuyoruz.
    defer cevap.Body.Close()
}

```

```
//Body'den gelen veriyi okuyoruz.  
sonuç, hata := io.ReadAll(cevap.Body)  
  
//Ya sev...  
if hata != nil {  
    panic(hata)  
}  
  
//İnsanlar byte dizisini okuyamadığı için  
//string tipine çeviriyoruz.  
fmt.Println(string(sonuç))  
}
```

Post İsteği

Post isteğini yaparken örnek bir json verisi göndermeyi görelim.

```
package main  
  
import (  
    "bytes"  
    "encoding/json"  
    "fmt"  
    "io"  
    "net/http"  
)  
  
func main() {  
    address := "https://orneksite.com/kisiler"  
    hamVeri := map[string]string{  
        "isim": "Kaan",  
        "soyisim": "Kuşcu",  
    }  
    jsonVeri, hata := json.Marshal(hamVeri)  
    //jsonVeri string tipine çevirildiğinde:  
    //{ "isim": "Kaan", "soyisim": "Kuşcu" }  
  
    if hata != nil {  
        panic(hata)  
    }  
  
    //Aşağıdaki gönderdiğimiz verinin json tipinde olduğunu
```

```

bildiriyoruz.
    sorgu, hata := http.Post(address, "application/json",
bytes.NewBuffer(jsonVeri))
    if hata != nil {
        panic(hata)
    }

    //sonucu sonuç değişkenine atayalım
    sonuç, hata := io.ReadAll(sorgu.Body)
    if hata != nil {
        panic(hata)
    }

    //Son olarak ekrana bastıralım.
    fmt.Println(string(sonuç))
}

```

Yukarıdaki örnekte json verimizi map ile oluşturarak post isteğinde kullandık. Map ile oluşturduğumuz json verisi esnek olmadığı için, yani yukarıdaki örnekte sadece string tipinde veri oluşturabildiğimiz için, bir de struct ile oluşturmayı görelim. Anonim bir struct oluşturup işimizi görebiliriz.

```

package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "io"
    "net/http"
)

func main() {
    address := "https://orneksite.com/kisiler"

    //struct içindeki değişken isimlerini büyük harfle
    //yazmazsak json'a çevirirken hata alırız.
    structVeri := struct {
        İsim      string `json:"isim"`    //json'da isim olacak
        Soyisim   string `json:"soyisim"` //json'da soyisim olacak
        Yaş       int    `json:"yas"`     //json'da yas olacak
    }{"Kaan", "Kuşcu", 23}
}

```

```
//struct'ımızı json'a çevirelim.  
jsonVeri, hata := json.Marshal(structVeri)  
//jsonVeri string tipine çevirildiğinde:  
//{"isim":"Kaan","soyisim":"Kuşcu","yas":23}  
  
if hata != nil {  
    panic(hata)  
}  
  
//Aşağıdaki gönderdiğimiz verinin json tipinde olduğunu  
bildiriyoruz.  
sorgu, hata := http.Post(address, "application/json",  
bytes.NewBuffer(jsonVeri))  
if hata != nil {  
    panic(hata)  
}  
  
//sonucu sonuç değişkenine atayalım  
sonuç, hata := io.ReadAll(sorgu.Body)  
if hata != nil {  
    panic(hata)  
}  
  
//Son olarak ekrana bastıralım.  
fmt.Println(string(sonuç))  
}
```



WebView

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.

zserge/webview repo'su webview/webview'a taşınmıştır.

Build işlemi sonrası aslında elimizde bir internet tarayıcısı olmuş oluyor. Bu tarayıcı üzerinden hazırlamış olduğumuz web sayfası görüntüleniyor. Frontend ve Backend arasındaki iletişimi ise **Bind** ile sağlıyoruz. Bu özelliği birazdan kodlar içerisinde açıklayacağım.

Sadece **Windows**, **GNU/Linux** ve **macOS** için uygulama geliştirebiliyoruz.

GNU/Linux üzerinde **gtk-webkit2**, macOS üzerinde **Cocoa/Webkit** ve Windows üzerinde **Edge** alt yapısını kullanıyor. Linux üzerinde çalışması için, gtk-webkit2 paketini yüklemeyi unutmayın. Bu detaylara bakacak olursak, Windows

üzerinde çalışırken Edge Browser'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/webview/webview
```

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

```
package main

//webview paketimizi içe aktaralım
import "github.com/webview/webview"

func main() {
    //debug değişkeninde debug modu açıyoruz.
    debug := true

    //yeni webview nesnesi oluşturduk.
    pencere := webview.New(debug)

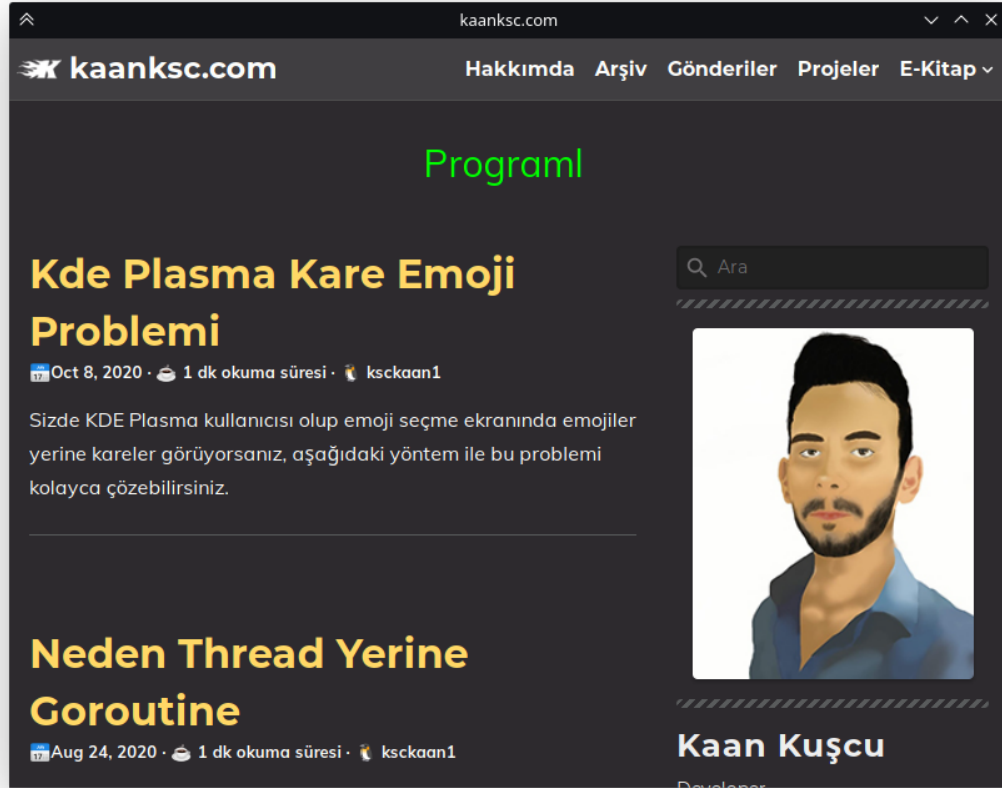
    //main fonksiyonunun sonunda pencerenin sonlanmasını
    //istedik.
    defer pencere.Destroy()

    //pencere başlığını belirttik.
    pencere.SetTitle("kaanksc.com")

    //penceremizin boyutunu belirttik.
    //(genişlik, uzunluk)
    pencere.SetSize(800, 600, webview.HintNone)
    //webview.HintNone ile normal bir pencere oluşturduk.

    //penceremizin yükleyeceği adresi belirtelim.
    pencere.Navigate("https://kaanksc.com")

    //son olarak penceremizi başlatalım.
    pencere.Run()
}
```

Oluşturulan pencerenin görünümü

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/webview/webview"
)

//anasayfayı yakalaması için bir yakalayıcı oluşturuyoruz.
func handler(w http.ResponseWriter, r *http.Request) {
    //anasayfada gösterilecek metnimiz.
    fmt.Fprintf(w, "Uygulamaya hoşgeldiniz!")
}
```

```

//sunucu oluşturan fonksiyonumuz.
func serverOlustur() {
    // anasayfanın handler fonksiyonu ile çalışmasını.
    //istiyoruz
    http.HandleFunc("/", handler)

    //sunucumuzun koşacağı portu belirtiyoruz.
    http.ListenAndServe(":5555", nil)
}
func main() {
    //webview penceremize eşzamanlı olarak sunucunun
    //koşturulmasını istiyoruz.
    go serverOlustur()

    //webview nesnesi oluşturuyoruz.
    pencere := webview.New(true)
    // New fonksiyonundaki true ifadesi debug modunda
    //çalışmasını sağlıyor. Sayfaya sağ tıklayarak
    //görebilirsiniz.

    //main fonksiyonun son işlemi olarak webview
    //penceresinin kaldırılmasını istiyoruz.
    defer pencere.Destroy()

    //penceremizin başlığını belirtiyoruz.
    pencere.SetTitle("Uygulamam")

    //penceremizin boyutunu belirtiyoruz.
    pencere.SetSize(400, 300, webview.HintNone)

    //hangi adresi yükleyeceğini belirtiyoruz.
    pencere.Navigate("http://localhost:5555")

    //son olarak penceremizi başlatıyoruz.
    pencere.Run()
}

```

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

tarafındaki webview 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.

Projemizin yapısı aşağıdaki gibi olsun.

```
.
├── klasor
│   ├── index.html
│   └── main.go
```

index.html dosyamız aşağıdaki gibi olsun.

```
<!DOCTYPE html>
<html lang="tr">
<head>
  <meta charset="UTF-8">
  <title>Bu kısma aslında gerek yok</title>
</head>
<body>
  <h1>Merhaba Dünya!</h1>
</body>
</html>
```

main.go dosyamız ise aşağıdaki gibi olsun.

```
package main

import (
    "net/http"

    "github.com/webview/webview"
)

//sunucu başlatan fonksiyonumuz
//aslında bunu main fonksiyonuna da
// "go http.ListenAndServe(":5555", nil)"
//şeklinde yazabilirsiniz.
func serverOlustur() {
    //dinlemek portu belirttik (5555)
    http.ListenAndServe(":5555", nil)
```

```

}
func main() {
    //Sunucumuzun hangi klasörden oluşacağını
    //belirtiyoruz. (klasor isimli klasörümüzden)
    klasor := http.FileServer(http.Dir("klasor/"))

    //anasayfayı klasor'e bağlayalım.
    http.Handle("/", http.StripPrefix("/", klasor))

    //eşzamanlı olarak sunucuyu başlatalım.
    go serverOlustur()

    //debug açık şekilde webview nesnesi oluşturalım.
    pencere := webview.New(true)

    //pencere başlığını girelim.
    pencere.SetTitle("Uygulamam")

    //pencere boyutunu ve normal pencere olacağını belirtelim.
    pencere.SetSize(600, 400, webview.HintNone)

    //Yukarıda başlattığımız sunucunun adresini girelim.
    pencere.Navigate("http://localhost:5555")

    //penceremizi başlatalım.
    pencere.Run()
}

```

Sıra geldi Backend (Golang) ve Frontend (Javascript) arasındaki iletişimi sağlamaya. Aşağıdaki işlemleri yukarıdaki klasör yapısında göre yapacağız. Yani bu şekilde:

```

.
├── klasor
│   └── index.html
└── main.go

```

Frontend'den Backend'e Veri Gönderme (JavaScript ==> Go)

Bu işlemi gerçekleştirebilmemiz için webview tarayıcısının frontend'deki sinyalleri dinlemesi gerekir. Golang tarafından dinlemek için **Bind()** fonksiyonunu kullanıyoruz.

Örnek bir **main.go** dosyası oluşturalım.

```
package main

import (
    "fmt"
    "net/http"

    "github.com/webview/webview"
)

func serverOlustur() {
    http.ListenAndServe(":5555", nil)
}

func main() {
    klasor := http.FileServer(http.Dir("klasor/"))
    http.Handle("/", http.StripPrefix("/", klasor))
    go serverOlustur()
    pencere := webview.New(true)
    pencere.SetTitle("Uygulamam")
    pencere.SetSize(600, 400, webview.HintNone)
    pencere.Navigate("http://localhost:5555")

    //Bind fonksiyonu ile dinleyici oluşturduk
    pencere.Bind("merhaba", func(isim string) {
        //Javascript'teki merhaba fonksiyonunu dinliyoruz.
        //Javascript'teki bu fonksiyona string bir değer girilecek.
        //Bu değeri de aşağıdaki gibi ekrana bastıracağız.
        fmt.Println("merhaba", isim)
    })
    pencere.Run()
}
```

Bind() fonksiyonunu pencere.Run()'dan önce yazmalıyız. Eğer sonra yazarsak, çalışma zamanı Run() fonksiyonu kapatılmadan Bind() fonksiyonuna gelmeyeceği için yazmamızın bir mantığı olmaz.

Sıra geldi websayfamızdaki kodları yazmaya.

klasor klasörümüzün içerisine **uygulama.js** dosyası oluşturalım.

```
.
├── klasor
│   └── index.html
```

```
└─ uygulama.js << burada
└─ main.go
```

index.html dosyamıza ise bir yazı kutusu ve buton ekleyelim.

Ek olarak da **uygulama.js** dosyamızı html etiketi olarak ekleyelim.

```
<!DOCTYPE html>
<html lang="tr">
<head>
  <meta charset="UTF-8">
  <title>Bu kısma aslında gerek yok</title>
</head>
<body>
  <h1>Merhaba Dünya!</h1><br>
  <input type="text" id="isim">
  <button id="gonder">Gönder</button>
  <script src="./uygulama.js"></script>
</body>
</html>
```

Oluşturduğumuz HTML elementlerine id'ler vererek, bunları Javascript'te kullanacağız.

uygulama.js dosyamızın içeriği ise aşağıdaki gibi olsun.

```
var yaziKutusu = document.querySelector("#isim")
//yazı kutusunun id'sini belirttik

var gonderButonu = document.querySelector("#gonder")
//Gönder butonunun id'sini belirttik

//Gönder butonuna tıklandığı zaman gerçekleşecek olaylar
gonderButonu.addEventListener("click",() => {
  //yazı kutusunun içindeki değeri isim değişkeni olarak
  belirledik.
  var isim = yaziKutusu.value;
  //merhaba fonksiyonuna isim değişkeninin değerini yolladık.
  merhaba(isim)
  //Farkettiyseniz javascript tarafında merhaba adında bir
  fonksiyon
  //oluşturmadık. Normalde tarayıcı buna hata verecektir.
  //Ama biz golang tarafındaki webview'e merhaba fonksiyonunu
  bildirdik
  //ve dinleyici olarak tanıttık.
})
```

Bu işlemler sonucunda uygulamamızı projemizin ana dizinindeyken `go run .` şeklinde başlatalım.

Yazı kutusuna isim girip **Gönder** butonuna bastığımızda konsol tarafında `merhaba isim` şeklinde bir çıktı görürüz.

Backend'den Frontend'e Veri Gönderme (Go ==> JavaScript)

Aslında burada yapacağımız olay bir JavaScript kodu çalıştırmak veya tetiklemek de denebilir. Tıpkı Developer Console'dan yaptığımız gibi..

Bu işlem için `Eval()` fonksiyonundan faydalanıyoruz. Örnek için yukarıdaki kodlarımıza devam edelim.

`merhaba` fonksiyonunu dinliyoruz demiştik. `Bind()` içerisinde komut satırına `isim` bastırmak yerine JavaScript konsoluna `isim` bastıralım.

`Bind()` fonksiyonumuzun içeriği şöyle olsun.

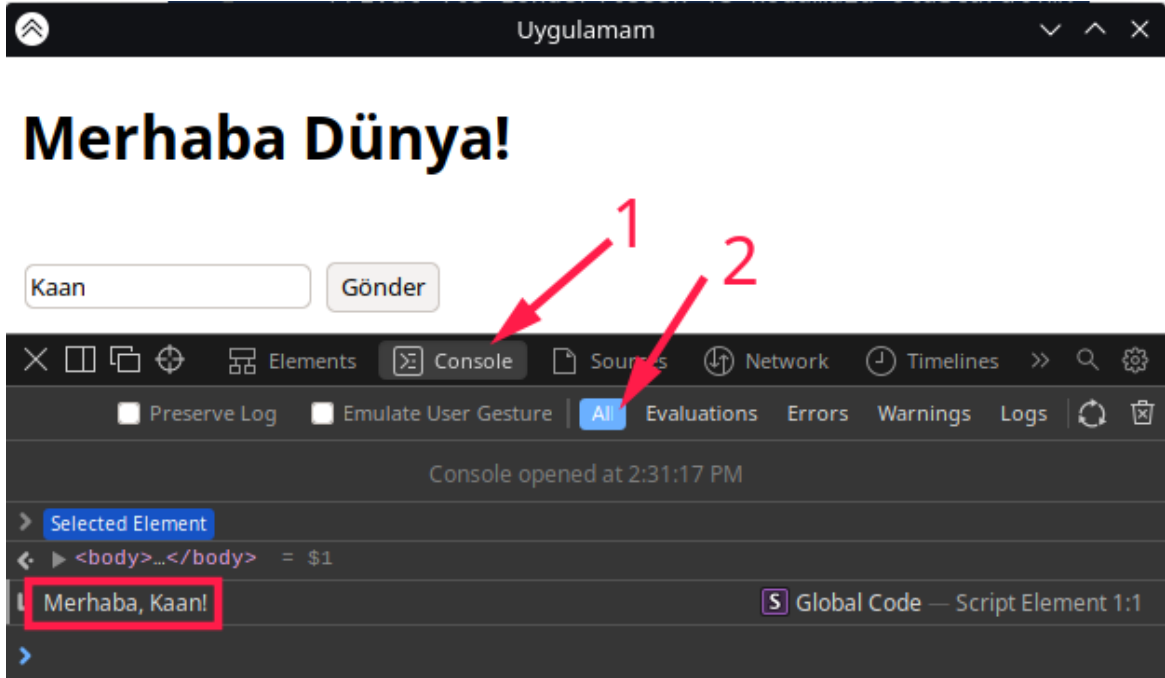
```
pencere.Bind("merhaba", func(isim string) {  
    //Eval ile gönderilecek js kodumuzu oluşturalım.  
    jsKodumuz := fmt.Sprintf("console.log('Merhaba, %s!')", isim)  
    //isim değişkeni %s yerinde gösterilecek.  
    pencere.Eval(jsKodumuz)  
})
```

Uygulamamızı çalıştırıp yazı kutusuna bir isim girdiğimizde Komut satırından şöyle bir çıktı alabiliriz:

<http://localhost:5555/:1:12>: CONSOLE LOG Merhaba, Kaan!

İsterseniz Webview içerisindeki Developer Tool ile de bakabiliriz.

Sayfaya sağ tıklayalım ve Inspect Element'e tıklayalım. Açılan bölümde **Console** sekmesine geçelim. All seçeneğinin seçili olduğundan emin olalım. Ve işte! log çıktımızın burada.



Developer Tool görünümü

Bu işlemler ile kolay bir şekilde Backend-Frontend arası iletişimi sağlayabilirsiniz.

Golang ile Webview'a Asenkron Müdahale Etme

Go tarafında asenkron işlemler için **Goroutine**'leri kullandığımızı belirtmiştik. Backend tarafında zaman alan bir işlemi goroutine ile asenkron bir işlem parçacığı haline getirmezsek. İşlem tamamlanana kadar webview pencremiz yanıt vermez. Yani herhangi bir şeye tıklayamayız. Bu yüzden uzun sürecek arkaplan işlemlerini **Webview**'e asenkron şekilde çalıştırmamız gerekir.

Örnek bir Bind() fonksiyonunda asenkron bir işlemi inceleyelim.

```
pencere.Bind("merhaba", func(isim string) {  
    //asekron çalışan bir anonim fonksiyon oluşturalım  
    go func(){  
        //3 saniye beklesin  
        time.Sleep(3*time.Second)  
        //ve ekrana JavaScript tarafından gelen
```



```

        //isim değişkenini bastırsın.
        fmt.Println(isim)
    }()
})

```

Yukarıdaki örnekte JavaScript tarafından çalıştırılacak merhaba fonksiyonunu dinliyoruz. Bu fonksiyon bize string tipinde bir değer iletiyor. Asenkron olarak anonim bir fonksiyon oluşturuyoruz ve içeriğinde 3 saniye beklemesini istiyoruz. Son olarak ekrana isim değişkenindeki değeri bastırıyoruz.

Eğer bu kodları asenkron olarak çalıştırmıyaydık, Webview penceremiz 3 saniye boyunca yaptığımız hiçbir işleme tepki vermeyecekti.

Şimdi buraya kadar karışık bir olay yok aslında. Asıl asenkron mantığı 3 saniye bekleyip çıktımızı JavaScript konsoluna bastırmak olacaktır. Aralarındaki asenkron olayı bu şekilde sağlanacaktır.

Normalde asenkron iki işlemin birinden diğerine ait olan bir işlemi yapmak için işaretçileri (*pointer*) kullanabiliriz. Fakat Webview nesnesine bir işaretçi atayamayız. Çünkü webview ile oluşturulan nesnemiz bir interface'tir.

Örnek bir deneme girişi 

```

package main

import (
    "fmt"
    "net/http"
    "time"

    "github.com/webview/webview"
)

func serverOlustur() {
    http.ListenAndServe(":5555", nil)
}

func main() {
    klasor := http.FileServer(http.Dir("klasor/"))
    http.Handle("/", http.StripPrefix("/", klasor))
    go serverOlustur()
}

```

```

pencere := webview.New(true)
pencere.SetTitle("Uygulamam")
pencere.SetSize(600, 400, webview.HintNone)
pencere.Navigate("http://localhost:5555")

pencere.Bind("merhaba", func(isim string) {
    //fonksiyonumuzu asenkron olarak başlatıyoruz
    go fonksiyonumuz(&pencere, isim)
    //parametre olarak pencere nesnesinin adresini verdik
    //aynı şekilde isim değişkenini de
})
pencere.Run()
}

func fonksiyonumuz(p *webview.WebView, isim string) {
    //3 saniye beklemesini istedik
    time.Sleep(3 * time.Second)
    //Eval ile gönderilecek JS kodumuzu hazırlayalım
    jsKodumuz := fmt.Sprintf("alert('%s')", isim)

    //İşte geldik Zurnanın zırt dediği yere (kullandığım deyme
    takılmayın ☹️)
    p.Eval(jsKodumuz)
    //Bu kısımda böyle bir kullanım yapamayacağımızı söylüyor
    olacak.
}

```

```

//İşte geldik Zurnanın zırt dediği yere (kullandığım deyme takılmayın ☹️)
p.Eval(jsKodumuz)    p.Eval undefined (type *webview.WebView is pointer to interface, not interface)
//Bu kısımda böyle bir kullanım yapamayacağımızı söylüyor olacak.

```

Hata mesajı

Şuana kadar gösterdiğim şeyler bir yanlış yapmayın diyeydi. Yukarıdaki yöntemi doğru değildir. Doğrusu `Dispatch()` fonksiyonunu kullanmaktır. Go kodlarımızın tamamını göreceğiz şekilde bir örnek verelim. Örnek kullanımı:

```

package main

import (
    "fmt"
    "net/http"
    "time"

    "github.com/webview/webview"
)

```

```

func serverOlustur() {
    http.ListenAndServe(":5555", nil)
}
func main() {
    klasor := http.FileServer(http.Dir("klasor/"))
    http.Handle("/", http.StripPrefix("/", klasor))
    go serverOlustur()
    pencere := webview.New(true)
    pencere.SetTitle("Uygulamam")
    pencere.SetSize(600, 400, webview.HintNone)
    pencere.Navigate("http://localhost:5555")

    pencere.Bind("merhaba", func(isim string) {
        //fonksiyonumuzu asenkron olarak başlatıyoruz
        go fonksiyonumuz(pencere, isim)
        //parametre olarak pencere nesnesini & olmadan verdik
    })
    pencere.Run()
}

func fonksiyonumuz(p webview.WebView, isim string) {
    //3 saniye beklemesini istedik
    time.Sleep(3 * time.Second)
    //Eval ile gönderilecek JS kodumuzu hazırlayalım
    jsKodumuz := fmt.Sprintf("alert('%s')", isim)

    //Dispatch fonksiyonu ile ana iş parçacığındaki webview
    //nesnesi ile işlem yapabiliriz.
    p.Dispatch(func() {
        p.Eval(jsKodumuz)
    })
}

```

Webview Kütüphanesindeki Diğer Fonksiyonlar

Destroy()

Webview penceremizi sonlandırır.

```
pencere.Destroy()
```

Terminate()

Pencerenin çalışmasını keser.

```
pencere.Terminate()
```

Eval()

Pencerede JavaScript kodu çalıştırmamızı sağlar.

```
pencere.Eval("alert('Merhaba!')")
```

Init()

Pencereye JavaScript kodu ilişitir. Eval() fonksiyonundan farkı ise sayfa değişse bile JavaScript kodu sayfada kalır. Eval() ile bir kere mahsus JavaScript kodu çalıştırılır. Init() fonksiyonunda sayfa yenilenince bile kod çalışırır.

```
pencere.Init("alert('Merhaba!')")
```

Navigate()

Webview penceresinin belirtilen adresi yüklemesini sağlar.

```
pencere.Navigate("https://www.google.com.tr")
```

Run()

Pencereyi başlatır.

```
pencere.Run()
```

SetSize()

Pencerenin boyutunu ve etkileşimini ayarlar.

```
//Genişlik, Uzunluk, Etkileşim  
pencere.SetSize(800, 600, webview.HintNone)  
//webview.HintNone ile normal,  
//webview.HintFixed ile boyutu değiştirilemeyen,  
//webview.HintMax ile en fazla solundaki büyüklükte,  
//webview.HintMin ile en az solundaki küçüklükte  
//bir pencere oluşturabiliriz.
```

SetTitle()

Pencerenin başlığını değiştirmemizi sağlar.

```
pencere.SetTitle("Uygulama Başlığı")
```

notify (Bildirim)

Bazen programımızda kullanıcının haberdar olması için bildirim özelliği gerekir. Bu özelliği `notify` paketi ile kazandırabiliriz.

```
go get github.com/martinlindhe/notify
```

Paketi indirdikten sonra, aşağıdaki kodları deneyebilirsiniz.

```
package main

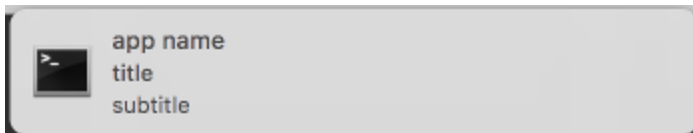
import "github.com/martinlindhe/notify"

func main() {
    // Bildirim Gösterme
    notify.Notify("Uygulama İsmi", "Başlık", "Açıklama",
        "icon/yolu.png")

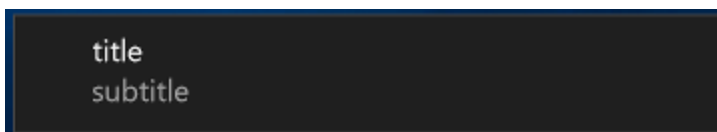
    // Sesli Bilgirim Gösterme
    notify.Alert("Uygulama İsmi", "Başlık", "Açıklama",
        "icon/yolu.png")
}
```



GNU/Linux (Ubuntu'da)



macOS / OSX 10.8+



Windows 10

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 Logosu

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](#)

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 Kurulumu

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 aratabilirsiniz. Fedora Linux sudo dnf install tinygo

Windows

Öncelikle şu anda 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 aşağıdakine 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.

Dinamik Değişkenler

Daha önceki bölümlerden de gördüğümüz üzere Go, dinamik atamayı desteklemiyor. Ama bunun elbette bir yöntemi var. Bunun için `interface`'den faydalanabiliriz.

Örnek:

```
package main

import (
    "fmt"
)

//dinamik atama yapabilmek için önce boş bir interface
oluşturalım
type dynamic interface{}

func main() {

    //x değişkenimizin tipini interface olarak belirleyelim
    var x dynamic

    //x'e integer tipinde değer atayalım
    x = 13

    //x'in tipini ve değerini ekrana bastıralım
    fmt.Printf("%T:%v\n", x, x) //int:13

    //Daha sonradan x'e string tipinde değer atayalım
    x = "selam"

    //x'in tipini ve değerini ekrana bastıralım.
    fmt.Printf("%T:%q\n", x, x) //string:"selam"
}
```

Yukarıdaki örnekte görüldüğü üzere `x` değişkenimize hangi tipte atama yaparsak, `x` değerinin tipine dönüşüyor.

Cobra Paketi ile Basit CLI Yazmak

Cobra'yı Yüklüyoruz

```
$ go get -u github.com/spf13/cobra/cobra
```

Yukarıdaki komutu yazarak yükleyebilirsiniz.

2. Projeyi Oluşturuyoruz

Şimdi Cobra CLI ile proje oluşturacağım. Projeyi aşağıdaki komutla oluşturabilirsiniz; `$ mkdir -p testApp && cd testApp $ cobra init --pkg-name github.com/your_user_name/testApp`

kısmını kendi GitHub kullanıcı adınızla değiştirmeyi unutmayın. Şu anda testApp klasörü içerisinde Cobra projesi oluşturulmuş olmalı.

Klasör hiyerarşisi aşağıdaki gibi olmalı:

.

└─ cmd

| └─ root.go

└─ main.go

1. Projeye Komut Ekleme

```
$ cobra add start
```

şeklinde ekliyoruz. Projeye bakarsanız cmd klasörü altına start.go dosyası oluşturulduğunu göreceksiniz. start.go dosyası içine bakarsak karşımıza şu çıkacaktır. Buradaki mantık Use kısmı bu komutun kullanımı bilgisini verir mesela

```
testApp start
```

gibi. Short kısmında ise bu oluşturduğumuz komutun kısa açıklamasıdır ve Long ise uzun açıklaması anlamına gelir.

```
var startCmd = &cobra.Command{
    Use:   "start",
    Short: "A brief description of your command",
    Long:  `A longer description that spans multiple lines and
likely contains examples
and usage of using your command. For example:
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("start called")
    },
}
func init() {
    rootCmd.AddCommand(startCmd)
}
```

Yukardaki kod örneğine göre Terminale, `>$ testApp start`

yazdığımızda start komutu çalışıyor.

2. Arguman Girdisi Almak

Cobra, girilen arguman sayısı veya kendi belirlediğiniz kıstaslara göre argumanları kontrol edip programınızın bu kontrol sonucuna göre bir çıktı vermesine imkan veren yapılara sahiptir. Arguman kullanımına basit bir örnek verelim;

```
func init() {
    rootCmd.AddCommand(Open)
}

var Open = &cobra.Command{
    Use:     "open ",
    Short:   "open your file in term",
    Args:    cobra.MaximumNArgs(1),
    Run:     func(cmd *cobra.Command, args []string) {
        name := strings.Join(args, " ")
        dosyaicerik, _ := ioutil.ReadFile(string(name))
        fmt.Println(string(dosyaicerik))
    },
}
```

Yukarda verdiğim örnekte

```
$ testApp open [dosya ismi]
```

şeklinde bir kullanım yaparak girdiğimiz dosya ismi yerine girilen dosyanın terminale basılmasını sağlamaktadır cat komutu gibi.

Uygulamayı Yüklemek

Projemizi kullanmak için hep go run ya da derlemek saçma olur bunun için proje dizinine,

```
$ go install
```

komutunu yazarak projeyi yükleyebilirsiniz.

Şimdi proje yüklemesini test etmek amaçlı aşağıdaki komutu gireim terminale çıktı alırsanız yüklenmiş demektir.

```
$ testApp
```

Go Geliştiricileri için Makefile

Golang ile yazılım geliştirirken **Makefile** teknolojisinde nasıl faydalanacağımızı göreceğiz. Gözümüzün korkmasına gerek yok, aşırı basit bir olay. Zaten herşeyi biliyorsunuz. Makefile sadece bir yöntemdir.

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'in genel mantığına baktığımızda komut satırı üzerinden yaptığımız işlemleri kısaltıyor.

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"
```

```
G00S=windows GOARCH=amd64 go build -o bin/main-
```

```
windows64.exe main.go
```

```
G00S=linux GOARCH=amd64 go build -o bin/main-linux64
```

```
main.go
```

```
G00S=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 ki Go'da kullanılacak diye bir kaide yok. Diğer programlama dillerinde veya komutlarınızı otomatize etmek istediğiniz zaman kullanabilirsiniz.

Derleme (Build) 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

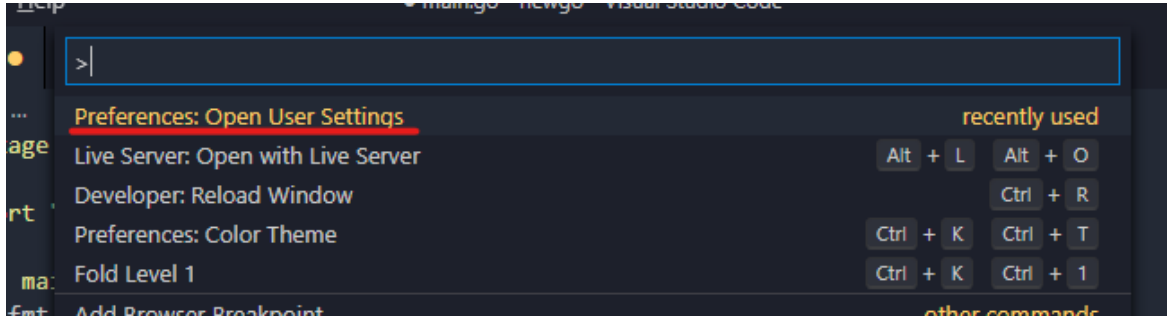
Visual Studio Code için Golang Özelleştirmeleri

Bu yazıda Visual Studio Code üzerinde Golang için kullanabileceğimiz özelleştirmelerden bahsedeceğiz. Bu özelleştirmeler sayesinde kod yazma deneyimimizi iyileştirebiliriz.

Canlı Hata Ayıklama

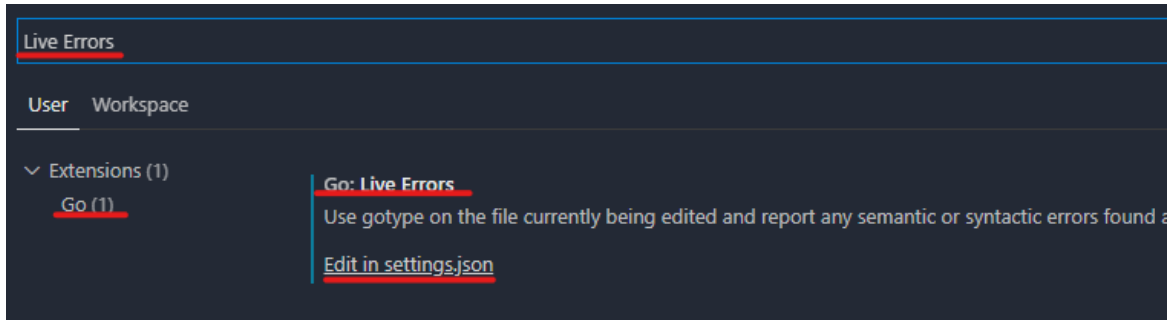
VSCoide üzerinde Go dili kodları yazarken farketmişsinizdir. Kodu yazarken hata ayıklamıyor. Sadece dosyayı kaydettiğimizde hata ayıklama işlemi yapıyor. Kod üzerinde canlı hata ayıklamayı aktif etmemiz gerekiyor. Bunun için;

CTRL + SHIFT + P tuşlarına beraber basarak, VSCoide komut bölümünü açalım. Bu kısma "Preferences: Open User Settings" yazalım ve çıkan ilk sonuca girelim.



Adım.1

Açılan Settings sekmesinde üst tarafta bulunan arama yapma kutusuna "Go: Live Errors" yazalım. Çıkan sonuçta "Edit in settings.json" bağlantısına tıklayalım.



Adım.2

```
"go.liveErrors": {  
  "enabled": false, //Burayı true yapalım.  
  "delay": 500 //tepki süresi  
}
```

Açılan editörde "go.liveErrors" anahtarının karşısında ayarlarımız var. "enabled" anahtarının değerini true yapalım. "delay" anahtarındaki değerde ise yazdıktan kaç milisaniye sonra hata ayıklama yapacağını belirtiyoruz. 500 (yarım saniye) normal bir değerdir.

Daha sonra bir .go dosyası oluşturalım veya hali hazırda .go dosyasını açalım. Açtığımız dosyanın içerisine birşeyler yazmaya çalıştığımızda VSCode'un sağ alt köşesinde bir uyarı verecektir. Bu uyarıda **Install** butonuna tıklayarak eklenti yükleme işlemini başlatalım. Bu eklenti canlı hata ayıklama yapmak için gereklidir. Yüklendiğinde **Output** sekmesinde aşağıdakine benzer bir sonuç alacaksınız.

```
Tools environment: GOPATH=C:Installing 1 tool at C:in  
module mode. gotype-live
```

```
Installing github.com/tylerb/gotype-live (C:-live.exe)  
SUCCEEDED
```

All tools successfully installed. You are ready to Go :).

Artık canlı hata ayıklama özelliğini kullanabilirsiniz.

```
main.go
main.go > {} main > main
1 package main
2
3 import "fmt"
4
5 func main() {
6
7
8
```

Canlı Hata Ayıklama

Go Yazarken Kullanabileceğiniz VSCode Eklentileri

ErrorLens

Bu eklenti ile kod yazarken eğer hata varsa alakalı satırın sağında hata mesajını görebilirsiniz. Bu eklenti Go için kullanmadan önce Go için canlı hata ayıklamayı açmanızı tavsiye ederim (*Yukarıda gösterdim*).

```
main.go > ...
1 package main
2
3 import "fmt" "fmt" imported but not used
4
5 func main() {
6     var bir string bir declared but not used
7     var iki float64 iki declared but not used
8
9     iki= "naber" cannot convert "naber" (untyped string constant) to float64
10 }
```

ErrorLens Eklentisi

Better Comments

Yorum satırlarını daha şık gösteren bir eklentidir. Tavsiye ederim. Satırın başına koyduğunuz işaret veya yazdığınız yazıya göre satırın rengi değişiyor.

```
/* Normal Yorum  
* Renkli Yorum  
! Uyarı Yorumu  
TODO To-Do Yorumu  
? Soru Yorumu  
*/  
fmt.Println("Merhaba Dünya!")
```

Better Comments Eklentisi

Sıkça Sorulan Sorular

<https://golang.org/doc/faq> adresindeki içerikten tercüme edilmiştir.

□ Kökenler

→ Projenin amacı nedir?

Go'nun başlangıcında, sadece on yıl önce, programlama dünyası bugünden farklıydı. Yazılım üretimi genellikle C ++ veya Java ile yapılıyordu. GitHub mevcut değildi. Çoğu bilgisayar henüz çok işlemcili değildi. Visual Studio ve Eclipse dışında, internette ücretsiz olarak birkaç IDE veya diğer üst düzey araçlar mevcuttu.

Bu arada, sunucu yazılımı geliştirmede birlikte çalıştığımız dilleri kullanmak için karşılaştığımız gereksiz karmaşıklıktan dolayı hayal kırıklığına uğradık. Bilgisayarlar, C, C ++ ve Java gibi diller ilk geliştirildiğinden beri çok daha hızlı hale gelmişti, ancak programlama eylemi tam olarak o kadar ilerlememişti. Ayrıca, çok işlemcinin evrensel hale geldiği açıktı, ancak çoğu dil onları verimli ve güvenli bir şekilde programlamak için çok az yardım sundu.

Bir adım geri gitmeye ve teknoloji geliştikçe önümüzdeki yıllarda yazılım mühendisliğine hangi önemli sorunların hâkim olacağını ve yeni bir dilin bunları nasıl çözebileceğini düşünmeye karar verdik. Örneğin, çok çekirdekli CPU'ların yükselişi, bir dilin bir tür eşzamanlılık veya paralellik için birinci sınıf destek sağlaması gerektiğini savundu. Buna mütakiben kaynak yönetimini büyük bir eşzamanlı programda izlenebilir hale getirmek için, çöp toplama veya

en azından bir tür güvenli otomatik bellek yönetimi gerekiyordu.

Bu düşünceler, Go'nun önce bir dizi fikir ve aranan veri, sonra da bir dil olarak ortaya çıktığı bir dizi tartışmaya yol açtı. Kapsamlı bir hedef, Go'nun çalışan programcıya takım oluşturmayı etkinleştirerek, kod biçimlendirme gibi sıradan görevleri otomatikleştirerek ve büyük kod tabanlarında çalışmanın önündeki engelleri kaldırarak daha fazla yardım etmesiydi.

Go'nun hedeflerinin ve bunlara nasıl ulaşıldığının veya en azından bunlara nasıl yaklaşıldığının çok daha kapsamlı bir açıklaması [Go at Google: Language Design in the Service of Software Engineering](#) adlı makalede mevcuttur.

→ Projenin tarihçesi nedir?

Robert Griesemer, Rob Pike ve Ken Thompson 21 Eylül 2007'de beyaz tahtada yeni bir dilin hedeflerini çizmeye başladılar. Birkaç gün içinde hedefler bir şeyler yapma planına ve bunun ne olacağına dair adil bir fikre yerleşti. Tasarım, ilgisiz çalışmaya paralel olarak yarı zamanlı devam etti. Ocak 2008'de Ken, fikirleri araştırmak için bir derleyici üzerinde çalışmaya başladı; çıktı olarak C kodunu üretti. Yıl ortasına gelindiğinde, dil tam zamanlı bir proje haline geldi ve bir üretim derleyicisini deneyecek kadar yerleşti. Mayıs 2008'de Ian Taylor, taslak şartnameyi kullanarak Go için bir GCC ön ucunu bağımsız olarak kullanmaya başladı. Russ Cox 2008'in sonlarında katıldı ve dilin ve kitaplıkların prototipten gerçeğe taşınmasına yardımcı oldu.

Go, 10 Kasım 2009'da halka açık bir açık kaynak projesi oldu. Topluluktan sayısız kişi fikirlere, tartışmalara ve kodlara katkıda bulundu.

Şu anda dünya çapında milyonlarca Go programcısı (gopher) var ve her gün daha fazlası var olacak. Go'nun başarısı beklentilerimizi çok aştı.

→ Gopher maskotu nereden geliyor?

Maskot ve logo, Plan 9 tavşanı [Glenda](#)'yı da tasarlayan [Renée French](#) tarafından tasarlandı. Gopher hakkında bir [blog yazısı](#), birkaç yıl önce bir [WFMU](#) tişört tasarımı için kullandığı birinden nasıl türetildiğini açıklıyor. Logo ve maskot, [Creative Commons Attribution 3.0 lisansı](#) kapsamındadır. Gopher, özelliklerini ve bunların doğru şekilde nasıl temsil edileceğini gösteren bir [model sayfasına](#) sahiptir. Model sayfası ilk olarak 2016 yılında Renée'nin Gophercon'da yaptığı bir [konuşmada](#) gösterildi. Kendine özgü özellikleri var; o Go gopher, herhangi bir yaşlı sincap değil.

→ Dilin adı Go mu Golang mı?

Dilin adı Go. “Golang” takma adının ortaya çıkmasının nedeni, web sitesinin bizim için mevcut olmayan go.org değil, [golang.org](#) olmasıdır. Yine de çoğu golang adını kullanır ve etiket olarak kullanışlıdır. Örneğin, dil için Twitter etiketi “#golang” dir. Dilin adı ne olursa olsun sadece Go'dur.

Bir yan not: [Resmi logo](#)nun iki büyük harfi olmasına rağmen, dil adı GO değil, Go yazılmıştır.

→ Neden yeni bir dil yarattınız?

Go, Google'da yaptığımız iş için mevcut diller ve ortamlarla ilgili hayal kırıklığından doğdu. Programlama çok zor hale geldi ve dil seçimi kısmen suçluydu. Etkili derlemeyi, verimli yürütmeyi veya programlama kolaylığını seçmek

gerekiyordu; üçü de aynı ana dilde mevcut değildi. C ++ veya daha az ölçüde Java yerine Python ve JavaScript gibi dinamik olarak yazılmış dillere geçerek güvenlik ve verimlilik üzerinde kolaylık seçebilmeliydi programcılar.

Endişelerimizde yalnız değildik. Yıllar sonra, programlama dilleri için oldukça sessiz bir manzaraya sahip olan Go, programlama dili geliştirmeyi tekrar aktif, neredeyse yaygın bir alan haline getiren birkaç yeni dilin (Rust, Elixir, Swift ve daha fazlası) ilklerinden biriydi.

Go, yorumlanmış, dinamik olarak yazılmış bir dilin programlama kolaylığını statik olarak yazılmış, derlenmiş bir dilin verimliliği ve güvenliğiyle birleştirmeye çalışarak bu sorunları ele aldı. Bu dil aynı zamanda ağa bağlı ve çok çekirdekli bilgi işlem desteğiyle modern olmayı hedefliyordu. Son olarak, Go ile çalışmanın hızlı olması amaçlanmıştı: Tek bir bilgisayarda büyük bir yürütülebilir dosya oluşturmak en fazla birkaç saniye sürecektir. Bu hedeflere ulaşmak için bir dizi dilbilimsel konuyu ele almak gerekiyor: ifade edici ama hafif bir yazım sistemi; eşzamanlılık ve çöp toplama; katı bağımlılık belirtimi; ve bunun gibi. Bunlar kütüphaneler veya araçlar tarafından iyi bir şekilde ele alınamaz; yeni bir dil aranıyordu.

[Go at Google](#) makalesi, Go dilinin tasarımının arkasındaki arka planı ve motivasyonu tartışmanın yanı sıra bu SSS'de sunulan yanıtların çoğu hakkında daha fazla ayrıntı sağlar.

→ **Go'nun ataları nelerdir? (veya esinlendikleri)**

Go, çoğunlukla Pascal / Modula / Oberon ailesinden (bildirimler, paketler) önemli girdiler ve ayrıca Newsqueak ve Limbo (eşzamanlılık) gibi Tony Hoare'nin CSP'sinden esinlenen dillerden bazı fikirlerle, C ailesinde (temel sözdizimi) bulunur. Ancak, her yerde yeni bir dildir. Her

açından dil, programcıların ne yaptığını ve programlamayı nasıl yapacağını, en azından yaptığımız programlama türünü, daha etkili, yani daha eğlenceli olması düşünülerek tasarlandı.

→ Tasarımda yol gösterici ilkeler nelerdir?

Go tasarlandığında, Java ve C ++, en azından Google'da, sunucuları yazmak için en yaygın kullanılan dillerdi. Bu dillerin çok fazla defter tutma (muhasibecilik) ve tekrarlama gerektirdiğini hissettik. Bazı programcılar, verimlilik ve yazım güvenliği pahasına Python gibi daha dinamik, akıcı dillere geçerek tepki gösterdi. Verimliliğe, güvenliğe ve akıcılığa tek bir dilde sahip olmanın mümkün olması gerektiğini hissettik.

Go, kelimenin her iki anlamında da yazma miktarını azaltmaya çalışır. Tasarımı boyunca dağınıklığı ve karmaşıklığı azaltmaya çalıştık. İleri tanımlamalar ve başlık (header) dosyaları yoktur; her şey tam olarak bir kez ilan edilir. Başlatma anlamlı, otomatik ve kullanımı kolaydır. Sözdizimi temiz ve anahtar kelimelerde hafiftir. Yazımda gereksizlik (`foo.Foo * myFoo = new (foo.Foo)`): = tanımla-ve-türet yapısı kullanılarak basit tür türetmesi ile azaltılır. Ve belki de en radikal olanı, tür hiyerarşisi yoktur: türler sadece ilişkilerdir. Bu basitleştirmeler, Go'nun etkileyici olmasına rağmen karmaşıklıktan ödün vermeden anlaşılabilir olmasını sağlar.

Bir diğer önemli ilke de kavramları ortogonal tutmaktır. Yöntemler her tür için uygulanabilir; yapılar (struct) verileri temsil ederken arayüzler (interface) soyutlamayı temsil eder. Ortogonallik, nesneler birleştğinde ne olduğunu anlamayı kolaylaştırır.

□ Kullanım

→ Google, Go'yu dahili olarak kullanıyor mu?

Evet. Go, Google içindeki üretimde yaygın olarak kullanılmaktadır. Bunun kolay bir örneği golang.org'un arkasındaki sunucudur. [Google App Engine](https://cloud.google.com/appengine/)'de bir üretim yapılandırmasında çalışan godoc belge sunucusudur.

Daha önemli bir örnek, Google'ın Chrome ikili (binary) dosyalarını ve apt-get (Debian paket yöneticisi oluyor kendisi) paketleri gibi diğer büyük yüklenebilir dosyaları sunan indirme sunucusu `dl.google.com`'dur.

Go, Google'da kullanılan tek dil değildir, ancak site güvenilirliği mühendisliği (SRE) ve büyük ölçekli veri işleme dahil olmak üzere birçok alan için anahtar dildir.

→ Başka hangi şirketler Go kullanıyor?

Go kullanımı dünya çapında artıyor, ancak hiçbir şekilde yalnızca bulut bilişim alanında değil. Go'da yazılan birkaç büyük bulut altyapı projesi Docker ve Kubernetes'tir, ancak çok daha fazlası vardır.

Yine de sadece bulut değil. Go Wiki, Go kullanan birçok şirketin bazılarını listeleyen ve düzenli olarak güncellenen bir [sayfa](#) içerir. Wiki'de ayrıca, dili kullanan şirketler ve projeler hakkındaki [başarı hikayeleri](#)ne bağlantılar içeren bir sayfa vardır.

→ Go programları C / C ++ programlarıyla bağlantılı mı?

C ve Go'yu aynı adres alanında birlikte kullanmak mümkündür, ancak bu doğal bir uyum değildir ve özel arayüz yazılımı gerektirebilir. Ayrıca, C'yi Go koduyla ilişkilendirmek, Go'nun sağladığı bellek güvenliği ve yığın yönetimi özelliklerinden mahrum kalmaktır. Bazen bir sorunu çözmek için C kitaplıklarını kullanmak kesinlikle gereklidir, ancak bunu yapmak her zaman saf Go kodunda bulunmayan bir risk unsuru getirir, bu yüzden dikkatli olun.

C'yi Go ile kullanmanız gerekiyorsa, nasıl devam edeceğiniz Go derleyici uygulamasına bağlıdır. Go ekibi tarafından desteklenen üç Go derleyici uygulaması vardır. Bunlar, GCC arka ucunu kullanan varsayılan derleyici gccgo ve LLVM altyapısını kullanan biraz daha az olgun bir gollvm'dir.

Gc, C'den farklı bir arama kuralı ve bağlayıcı kullanır ve bu nedenle doğrudan C programlarından veya tam tersi şekilde çağrılmaz. [Cgo](#) programı, C kitaplıklarının Go kodundan güvenli bir şekilde çağrılmasına izin veren bir "yabancı işlev arabirimi" mekanizması sağlar. SWIG C ++ kitaplıklarına erişme yeteneğini genişletir.

Gccgo ve gollvm ile cgo ve SWIG'i de kullanabilirsiniz. Geleneksel bir API kullandıklarından, büyük bir dikkatle, bu derleyicilerden gelen kodu doğrudan GCC / LLVM-derlenmiş C veya C ++ programlarına bağlamak da mümkündür. Ancak, bunu güvenli bir şekilde yapmak, ilgili tüm diller için çağrı kurallarının anlaşılmasını ve ayrıca Go'dan C veya C ++ çağrılırken yığın sınırlarının dikkate alınmasını gerektirir.

→ Hangi IDE'lerin Go desteği var?

Go projesi özel bir IDE içermez, ancak dil ve kitaplıklar, kaynak kodunu analiz etmeyi kolaylaştıracak şekilde tasarlanmıştır. Sonuç olarak, en iyi bilinen editörler ve IDE'ler, doğrudan veya bir eklenti aracılığıyla Go destekler.

İyi Go desteğine sahip tanınmış IDE'lerin ve editörlerin listesi Emacs, Vim, VSCode, Atom, Eclipse, Sublime, IntelliJ (Goland adlı özel bir varyant aracılığıyla) ve daha fazlasını içerir. En sevdiğiniz ortam, Go'da programlama için üretken bir ortam olabilir.

→ Go, Google'ın protokol tamponlarını (buffers) destekliyor mu?

Ayrı bir açık kaynak projesi, gerekli derleyici eklentisini ve kitaplığı sağlar. github.com/golang/protobuf adresinde mevcuttur.

→ Go ana sayfasını başka bir dile çevirebilir miyim?

Kesinlikle. Geliştiricileri kendi dillerinde Go Language siteleri oluşturmaya teşvik ediyoruz. Ancak, sitenize Google logosunu veya markasını eklemeyi seçerseniz (golang.org'da görünmez), www.google.com/permissions/guidelines.html adresindeki yönergelere uymanız gerekecektir.

□ Tasarım

→ Go çalışma zamanına (runtime) sahip mi?

Go, her Go programının bir parçası olan, çalışma zamanı adı verilen kapsamlı bir kitaplığa sahiptir. Çalışma zamanı kitaplığı çöp toplama, eşzamanlılık, yığın yönetimi ve Go dilinin diğer kritik özelliklerini uygular. Dil için daha merkezi olmasına rağmen, Go'nun çalışma zamanı C kitaplığı olan `libc`'ye benzer.

Bununla birlikte, Go'nun çalışma zamanının Java çalışma zamanı tarafından sağlananlar gibi bir sanal makine içermediğini anlamak önemlidir. Go programları, yerel makine koduna (veya bazı varyant uygulamaları için JavaScript veya WebAssembly) önceden derlenir. Bu nedenle, terim genellikle bir programın çalıştığı sanal ortamı tanımlamak için kullanılsa da, Go'da "çalışma zamanı" kelimesi yalnızca kritik dil hizmetleri sağlayan kitaplığa verilen addır.

→ Unicode tanımlayıcılarından n'aber?

Go'yu tasarlarken aşırı ASCII merkezli olmadığından emin olmak istedik. Bu da tanımlayıcıların alanını 7 bitlik ASCII'nin sınırlarından genişletmek anlamına geliyordu. Go kuralı — tanımlayıcı karakterler Unicode tarafından tanımlandığı gibi harf veya rakam olmalıdır — anlaşılması ve uygulanması kolaydır, ancak kısıtlamaları vardır. Örneğin, karakterleri birleştirmek tasarım tarafından hariç tutulur ve bu, Devanagari gibi bazı dilleri hariç tutar.

Bu kuralın talihsiz bir sonucu daha var. Dışa aktarılan bir tanımlayıcının bir büyük harfle başlaması gerektiğinden, bazı dillerde karakterlerden oluşturulan tanımlayıcılar tanım gereği dışa aktarılamaz. Şimdilik tek çözüm, açıkça tatmin edici olmayan X 日本語 gibi bir şey kullanmaktır.

Dilin en eski sürümünden bu yana, diğer yerel dilleri kullanan programcıları barındırmak için tanımlayıcı alanının en iyi şekilde nasıl genişletilebileceği konusunda önemli ölçüde düşünülmüştür. Tam olarak ne yapılacağı aktif bir tartışma konusu olmaya devam ediyor ve dilin gelecekteki bir versiyonu tanımlayıcı tanımında daha liberal olabilir. Örneğin, Unicode organizasyonunun tanımlayıcılar için önerilerinden bazı fikirleri benimseyebilir. Ne olursa olsun, Go'nun en sevdiğimiz özelliklerinden biri olan harf

durumunun tanımlayıcıların görünürlüğüne belirleme şeklini korurken (veya belki genişletirken) uyumlu bir şekilde yapılmalıdır.

Şimdilik, daha sonra programları bozmadan genişletilebilecek basit bir kuralımız var. Belirsiz tanımlayıcıları kabul eden bir kuraldan kesinlikle kaynaklanabilecek hataları önleyen bir kural.

→ Go neden X özelliğine sahip değil?

Her dil yeni özellikler içerir ve birinin en sevdiği özelliği atlar. Go, programlamanın mutluluğu, derleme hızı, kavramların ortogonalitesi ve eşzamanlılık ve çöp toplama gibi özellikleri destekleme ihtiyacı göz önünde bulundurularak tasarlandı. En sevdiğiniz özellik, uymadığından, derleme hızını veya tasarımın netliğini etkilediği veya temel sistem modelini çok zorlaştıracığı için eksik olabilir.

Go'nun X özelliğinin eksik olması sizi rahatsız ediyorsa, lütfen bizi affedin ve Go'nun sahip olduğu özellikleri araştırın. X'in eksikliğini ilginç yollarla telafi ettiklerini görebilirsiniz.

→ Go'da neden jenerik tipler yok?

Jenerikler bir noktada eklenebilir. Bazı programcıların yaptığını bilsek de, onlar için bir aciliyet hissetmiyoruz.

Go, zaman içinde bakımı kolay olacak sunucu programları yazmak için bir dil olarak tasarlandı. (Daha fazla arka plan için [bu makale](#)ye bakın.) Tasarım, ölçeklenebilirlik, okunabilirlik ve eşzamanlılık gibi şeylere odaklandı. Polimorfik programlama o zamanlar dilin hedefleri için gerekli görünmüyordu ve bu yüzden basitlik için dışarıda bırakıldı.

Dil artık daha olgundur ve bir tür genel programlamayı düşünmek için alan vardır. Ancak, bazı uyarılar var.

Jenerikler kullanışlıdır, ancak tip sistem ve çalışma süresinde karmaşıklık açısından bir maliyete sahiptirler. Karmaşıklığa orantılı değer veren bir tasarım henüz bulamadık, ancak üzerinde düşünmeye devam ediyoruz. Bu arada, Go'nun yerleşik haritaları (map) ve dilimlerinin (slice) yanı sıra konteynerler oluşturmak için boş arayüzü kullanma yeteneği (açık kutudan çıkarma ile), çoğu durumda kod yazmanın mümkün olduğu anlamına gelir. Bu, daha az sorunsuz olsa da jeneriklerin sağlayacağı şeyi yapar.

Konu açık kalır. Go için iyi bir jenerik çözüm tasarlamaya yönelik önceki birkaç başarısız girişime bir göz atmak için [bu öneriye](#) bakın.

→ Neden Go'da Exceptions yok?

Try-catch-final deyiminde olduğu gibi bir kontrol yapısına istisnaların birleştirilmesinin kıvrımlı kodla sonuçlandığına inanıyoruz. Ayrıca, programcıları, bir dosyayı açamama gibi çok fazla sıradan hatayı istisnai olarak etiketlemeye teşvik etme eğilimindedir.

Go farklı bir yaklaşım sergiliyor. Düz hata işleme için, Go'nun çoklu değer dönüşleri, dönüş değerini aşırı yüklemeyen bir hatayı bildirmeyi kolaylaştırır. [Go'nun diğer özellikleriyle birlikte kanonik bir hata türü](#), hata işlemeyi keyifli hale getirir ancak diğer dillerdekinden oldukça farklıdır.

Go ayrıca, gerçekten istisnai koşullardan sinyal almak ve kurtarmak için birkaç yerleşik işleve sahiptir. Kurtarma mekanizması, yalnızca bir hatadan sonra yıkılan bir işlevin durumunun bir parçası olarak yürütülür; bu, felaketi ele almak için yeterlidir, ancak ekstra kontrol yapıları

gerektirmez ve iyi kullanıldığında temiz hata işleme koduyla sonuçlanabilir. Ayrıntılar için [Defer, Panic ve Recover makalesine](#) bakın.

Ayrıca, [Hatalar blog gönderisi](#)nin bir Go'da hataları temiz bir şekilde işleme yaklaşımı, hataların sadece değerler olduğundan, Go'nun tam gücünün hata işlemede kullanılabileceğini gösterir.

→ Go'da neden assertions (iddialar) yok?

Go, assertions sağlamaz. İnkâr edilemez derecede kullanışlıdır, ancak bizim deneyimlerimiz, programcıların bunları doğru hata işleme ve raporlama hakkında düşünmekten kaçınmak için koltuk değneği olarak kullanmasıdır. Doğru hata işleme, sunucuların önemli olmayan bir hatadan sonra çökmek yerine çalışmaya devam etmesi anlamına gelir. Doğru hata raporlama, hataların doğrudan ve yerinde olduğu anlamına gelir ve programcıyı büyük bir çökme izini yorumlamaktan kurtarır. Hataları gören programcı koda aşına olmadığında kesin hatalar özellikle önemlidir.

Bunun bir çekişme noktası olduğunu anlıyoruz. Go dilinde ve kütüphanelerde modern uygulamalardan farklı birçok şey vardır, çünkü bazen farklı bir yaklaşımın denemeye değer olduğunu düşünüyoruz.

→ Neden CSP fikirleri üzerine eşzamanlılık inşa etmelisiniz?

Eşzamanlılık ve çok iş parçacıklı programlama, zaman içinde zorluklarla ilgili bir ün geliştirmiştir. Bunun kısmen [pthreads](#) gibi karmaşık tasarımlardan ve kısmen de mutexler, koşul değişkenleri ve bellek engelleri gibi düşük seviyeli ayrıntılara aşırı vurgu yapılmasından kaynaklandığına

inaniyoruz. Daha yüksek seviyeli arayüzler, kapakların altında hala mutexler ve benzeri şeyler olsa bile çok daha basit bir kod sağlar.

Eşzamanlılık için üst düzey dil desteği sağlamanın en başarılı modellerinden biri, Hoare'nin İletişim Sıralı Süreçlerinden veya CSP'den gelir. Occam ve Erlang, CSP'den kaynaklanan iki iyi bilinen dildir. Go'nun eşzamanlılık ilkelleri, ana katkısı birinci sınıf nesneler olarak güçlü kanal kavramı olan aile ağacının farklı bir bölümünden türemiştir. Daha önceki birkaç dil ile ilgili deneyimler, CSP modelinin bir prosedürel dil çerçevesine çok iyi uyduğunu göstermiştir.

→ Neden Thread yerine Goroutine?

Goroutinler, eşzamanlılığın kullanımını kolaylaştırmanın bir parçasıdır. Bir süredir ortalıkta olan fikir, işlevleri - koroutinleri - bağımsız olarak bir dizi iş parçacığına çoğaltmaktır. Bir coroutine bloke edildiğinde, örneğin bir bloke edici sistem çağrısını çağırarak bloke edildiğinde, çalışma zamanı aynı işletim sistemi iş parçacığındaki diğer koroutinleri otomatik olarak farklı, çalıştırılabilir bir iş parçacığına taşır, böylece bunlar engellenmez. Programcı bunların hiçbirini görmez, önemli olan budur.

Gorutinler dediğimiz sonuç çok ucuz olabilir: sadece birkaç kilobayt olan yığın hafızasının ötesinde çok az ek yüke sahiptirler. Yığınları küçültmek için Go'nun çalışma zamanı yeniden boyutlandırılabilir, sınırlı yığınlar kullanır. Yeni basılmış bir goroutine birkaç kilobayt verilir ve bu neredeyse her zaman yeterlidir. Değilse, çalışma zamanı yığını otomatik olarak depolamak için belleği büyütür (ve küçültür), böylece birçok gorutinün mütevazı bir bellek miktarında yaşamasına izin verir. CPU ek yükü, işlev çağrısı başına yaklaşık üç ucuz talimatın ortalamasını alır. Bu aynı adres alanında yüz binlerce gorutin oluşturmak için pratik.

Gorutinler sadece iş parçacıkları olsaydı, sistem kaynakları çok daha az sayıda tükenirdi.

→ Map işlemleri neden atomik olarak tanımlanmıyor?

Uzun tartışmalardan sonra, haritaların tipik kullanımının birden fazla gorutinden güvenli erişim gerektirmediğine karar verildi ve bu durumlarda, map muhtemelen zaten senkronize edilmiş daha büyük bir veri yapısının veya hesaplamanın bir parçasıydı. Bu nedenle, tüm map işlemlerinin bir mutex yakalamasını gerektirmek çoğu programı yavaşlatır ve birkaçına güvenlik ekler. Kontrolsüz map erişiminin programı çökertebileceği anlamına geldiğinden, bu kolay bir karar değildi.

Dil, atomik map güncellemelerini engellemez. Güvenilmeyen bir programı barındırırken olduğu gibi gerektiğinde, uygulama map erişimini birbirine bağlayabilir.

Map erişimi, yalnızca güncellemeler yapılırken güvenli değildir. Tüm gorutinler yalnızca okuduğu - range için döngü (for) kullanarak yineleme dahil map'teki öğeleri aradığı ve öğelere atayarak veya silme işlemleri yaparak map'i değiştirmedeği sürece, map'e eşzamanlı olarak erişimleri güvenlidir.

Map kullanımının düzeltilmesine yardımcı olarak, bazı Dil uygulamaları, bir map eşzamanlı yürütmeye güvenli olmayan bir şekilde değiştirildiğinde çalışma zamanında otomatik olarak rapor veren özel bir denetim içerir.

→ Dil değişikliği kabul edecek misiniz?

İnsanlar genellikle dilde iyileştirmeler yapılmasını önerir - [posta listesi](#) bu tür tartışmaların zengin bir geçmişini içerir -

ancak bu deęişikliklerin çok azı kabul edilmiştir. Go açık kaynaklı bir proje olmasına rağmen, dil ve kitaplıklar, en azından kaynak kodu düzeyinde mevcut programları bozan deęişiklikleri önleyen bir uyumluluk vaadi ile korunmaktadır (programların güncel kalması için ara sıra yeniden derlenmesi gerekebilir). Öneriniz Go 1 spesifikasyonunu ihlal ederse, deęeri ne olursa olsun fikri dikkate alamayız bile.

Go'nun gelecekteki büyük bir sürümü Go 1 ile uyumsuz olabilir, ancak bu konuyla ilgili tartışmalar daha yeni başladı ve kesin olan bir şey var: süreçte ortaya çıkan bu tür uyumsuzluklar çok az olacak. Dahası, uyumluluk vaadi, eski programların bu durum ortaya çıktığında adapte olması için ileriye dönük otomatik bir yol sağlamaya bizi teşvik ediyor. Teklifiniz Go 1 spesifikasyonu ile uyumlu olsa bile, Go'nun tasarım hedeflerinin ruhuna uygun olmayabilir. [Google'da Go: Yazılım Mühendisliği Hizmetinde Dil Tasarımı](#), Go'nun kökenlerini ve tasarımının arkasındaki motivasyonu açıklıyor.

□ Türler

→ Go nesne yönelimli bir dil midir?

Evet ve hayır. Go'nun türleri ve yöntemleri olmasına ve nesneye yönelik bir programlama stiline izin vermesine rağmen, tür hiyerarşisi yoktur. Go'daki "interface" kavramı, kullanımının kolay ve bazı açılardan daha genel olduğuna inandığımız farklı bir yaklaşım sağlar. Alt sınıflara benzer - ancak aynı olmayan - bir şey sağlamak için diğer türlere türleri gömmenin yolları da vardır. Dahası, Go'daki yöntemler C ++ veya Java'dakinden daha geneldir: düz, "unboxed" tamsayılar gibi yerleşik türler dahil her tür veri için tanımlanabilirler. Yapılar (sınıflar) ile sınırlı değildirler.

Ayrıca, bir tür hiyerarşisinin olmaması, Go'daki “nesnelerin” C ++ veya Java gibi dillerden çok daha hafif hissetmesini sağlar.

→ Yöntemlerin dinamik gönderimini nasıl edinebilirim?

Yöntemleri dinamik olarak göndermenin tek yolu bir interface kullanmaktır. Bir yapı veya başka herhangi bir somut türdeki yöntemler her zaman statik olarak çözümlenir.

→ Neden tür mirası (kalıtım) yok?

Nesne yönelimli programlama, en azından en iyi bilinen dillerde, türler arasındaki ilişkilerin, genellikle otomatik olarak türetilen ilişkilerin çok fazla tartışılmasını içerir. Go farklı bir yaklaşım benimser.

Go'da programcının önceden iki türün ilişkili olduğunu bildirmesini istemek yerine, Go'da bir tür, yöntemlerinin bir alt kümesini belirten herhangi bir interface'i otomatik olarak tatmin eder. Muhasebe tutmayı azaltmanın yanı sıra, bu yaklaşımın gerçek avantajları vardır. Türler, geleneksel çoklu kalıtımın karmaşıklıkları olmaksızın birçok arabirimi aynı anda karşılayabilir. Arayüzler çok hafif olabilir — bir hatta sıfır metotlu bir arayüz faydalı bir kavramı ifade edebilir. Arayüzler, yeni bir fikir ortaya çıkarsa veya test için orijinal tiplere açıklama yapmadan eklenebilir. Türler ve arabirimler arasında açık ilişkiler olmadığından, yönetilecek veya tartışılacak tür hiyerarşisi yoktur. Bu fikirleri benzer bir şey inşa etmek için kullanmak mümkündür. Örneğin, `fmt.Fprintf`'in sadece bir dosyaya değil herhangi bir çıktıya biçimlendirilmiş yazdırmayı nasıl sağladığını veya `bufio` paketinin dosya `G / Ç`'sinden nasıl tamamen ayrı

olabileceğini veya görüntü paketlerinin sıkıştırılmış görüntü dosyalarını nasıl oluşturduğunu görün.

Tüm bu fikirler, tek bir yöntemi (Yazma) temsil eden tek bir arayüzden (io.Writer) kaynaklanır. Ve bu sadece yüzeyi çizer. Go'nun arayüzlerinin, programların nasıl yapılandırıldığı üzerinde derin bir etkisi vardır. Alışmak biraz zaman alır, ancak bu örtük tip bağımlılığı Go ile ilgili en verimli şeylerden biridir.

→ **len() neden bir metod değilde fonksiyondur?**

Bu konuyu tartıştık, ancak pratikte işlevler iyi olduğu için len ve arkadaşlarını uygulamaya koymaya karar verdik ve temel türlerin interface'i (Go türü anlamında) hakkındaki soruları karmaşıktırmadı.

→ **Go neden method ve operatör overloading desteklemiyor?**

Tür eşleştirmesi de yapması gerekmiyorsa method gönderimi basitleştirilmiştir. Diğer dillerle edindiğimiz deneyimler bize, aynı isimde ancak farklı imzalara sahip çeşitli yöntemlere sahip olmanın bazen yararlı olduğunu, ancak pratikte kafa karıştırıcı ve kırılgan olabileceğini söyledi. Yalnızca isme göre eşleştirme ve türlerde tutarlılık gerektirme, Go'nun tür sisteminde büyük bir basitleştirici karardı. Operatörün aşırı yüklenmesi ile ilgili olarak, mutlak bir gereklilikten daha fazla bir kolaylık gibi görünüyor. Yine de, onsuz işler daha basit.

→ **Go neden “uygular” (implements) tanımlamalarına sahip değil?**

Bir Go türü, o arayüzün metodlarını uygulayarak bir arayüzü tatmin eder, başka bir şey değil. Bu özellik, arayüzlerin mevcut kodu değiştirmeye gerek kalmadan tanımlanmasına ve kullanılmasına izin verir. Kaygıların ayrılmasını teşvik eden ve kodun yeniden kullanımını geliştiren ve kod geliştikçe ortaya çıkan kalıplar üzerine inşa etmeyi kolaylaştıran bir tür [yapısal yazım](#) sağlar. Arayüzlerin anlamsallığı, Go'nun çevik ve hafif yapısının ana nedenlerinden biridir. Daha fazla ayrıntı için [tür mirası hakkındaki soru](#)ya bakın.

→ Türümün bir arayüzü karşıladığını nasıl garanti edebilirim?

Derleyiciden, uygun şekilde, T için sıfır değerini veya T'ye işaretçi kullanarak bir atama yaparak T türünün arayüzü I uygulayıp uygulamadığını kontrol etmesini isteyebilirsiniz:

```
type T struct{}
var _ I = T{}           // T'nin I'ya tanımlandığını doğrulayın.
var _ I = (*T)(nil)    // *T'nin I'ya tanımlandığını doğrulayın.
```

T (veya * T, buna göre) I'ya uygulamazsa, hata derleme zamanında yakalanacaktır. Bir arayüzün kullanıcılarının onu uyguladıklarını açıkça beyan etmelerini isterseniz, arayüzün yöntem kümesine açıklayıcı bir ada sahip bir yöntem ekleyebilirsiniz. Örneğin:

```
type Fooer interface {
    Foo()
    ImplementsFooer()
}
```

Bir tür daha sonra bir Fooer olmak için ImplementsFooer yöntemini uygulamalı, gerçeği açıkça belgelemeli ve [go doc](#)'un çıktısında duyurmalıdır.

```
type Bar struct{}  
func (b Bar) ImplementsFooer() {}  
func (b Bar) Foo() {}
```

Çoğu kod, arayüz fikrinin faydasını sınırladıkları için bu tür kısıtlamaları kullanmaz. Yine de bazen benzer arayüzler arasındaki belirsizlikleri çözmek için gereklidirler.

→ T türü neden Equal arayüzünü karşılamıyor?

Bu basit arayüzü, kendisini başka bir değerle karşılaştırabilen bir nesneyi temsil edecek şekilde düşünün:

```
type Equaler interface {  
    Equal(Equaler) bool  
}
```

ve bu tür, T:

```
type T int  
func (t T) Equal(u T) bool { return t == u } // Equaler'ı  
karşılamıyor
```

Bazı polimorfik tip sistemlerde benzer durumdan farklı olarak, T, Equaler'ı uygulamaz. T.Equal'ın bağımsız değişken türü T'dir, tam anlamıyla gerekli olan Equaler türü değildir.

Go'da tip sistemi Equal argümanını desteklemez; Bu, Equaler'ı uygulayan T2 türünde gösterildiği gibi programcının sorumluluğundadır:

```
type T2 int  
func (t T2) Equal(u Equaler) bool { return t == u.(T2) }  
//Equaleri karşılar
```

Bu bile diğer tür sistemler gibi değildir, çünkü Go'da Equaler'ı karşılayan herhangi bir tür argüman olarak T2.Equal'a aktarılabilir ve çalışma zamanında argümanın T2 türünde olup olmadığını kontrol etmeliyiz. Bazı diller bu

garantiyi derleme zamanında verir. İlgili bir örnek diğer tarafa gider:

```
type Opener interface {  
    Open() Reader  
}  
  
func (t T3) Open() *os.File
```

Go'da T3, başka bir dilde olsa da Opener'ı karşılamıyor. Go'nun tip sisteminin bu gibi durumlarda programcı için daha az şey yaptığı doğru olsa da, alt tipten olmaması arayüz memnuniyeti ile ilgili kuralları belirtmeyi çok kolaylaştırır: fonksiyonun adları ve imzaları tam olarak arayüzünkiler mi?

Go kuralının verimli bir şekilde uygulanması da kolaydır. Bu avantajların, otomatik tip promosyon eksikliğini telafi ettiğini düşünüyoruz. Bir gün, bir tür polimorfik yazım benimsemesi durumunda, bu örneklerin fikrini ifade etmenin ve ayrıca bunların statik olarak kontrol edilmesini sağlamanın bir yolu olacağını umuyoruz.

→ Bir []T'yi bir []interface'e dönüştürebilir miyim?

Direkt olarak değil. İki tür bellekte aynı temsile sahip olmadığından dil belirtimine göre buna izin verilmez. Öğeleri tek tek hedef dilime kopyalamak gerekir. Bu örnek, bir int dilimini bir interface{} dilimine dönüştürür:

```
t := []int{1, 2, 3, 4}  
s := make([]interface{}, len(t))  
for i, v := range t {  
    s[i] = v  
}
```


→ T1 ve T2 aynı temel türe sahipse [] T1'i [] T2'ye dönüştürebilir miyim?

Bu kod örneğinin bu son satırı derlenmiyor.

```
type T1 int
type T2 int
var t1 T1
var x = T2(t1) // Tamam
var st1 []T1
var sx = ([]T2)(st1) // tamam değil
```

Go'da türler, her adlandırılmış türün (muhtemelen boş) bir yöntem kümesine sahip olması nedeniyle metodlara yakından bağlıdır. Genel kural, dönüştürülen türün adını değiştirebilmeniz (ve dolayısıyla muhtemelen yöntem kümesini değiştirebilmeniz), ancak bileşik türdeki öğelerin adını (ve yöntem kümesini) değiştirememenizdir. Go, tür dönüşümleri konusunda açık olmanızı gerektirir.

→ Nil hata değerim neden nil'e eşit değil?

Kapakların altında, arayüzler iki öge olarak uygulanır; bir T türü ve bir V değeri V, int, struct veya işaretçi gibi somut bir değerdir, hiçbir zaman arabirim değildir ve T türüne sahiptir. Örneğin, int değeri 3 bir arabirimde, ortaya çıkan arabirim değeri şematik olarak (T = int, V = 3) olur. V değeri, aynı zamanda arayüzün dinamik değeri olarak da bilinir, çünkü belirli bir arayüz değişkeni, programın yürütülmesi sırasında farklı V değerlerini (ve karşılık gelen T tiplerini) tutabilir. Bir arabirim değeri, yalnızca V ve T'nin her ikisi de ayarlanmamışsa sıfırdır (T = nil, V ayarlanmamış). Özellikle, bir sıfır arabirim her zaman bir sıfır türünü tutacaktır. Bir arabirim değerinin içinde *int türünde bir sıfır gösterici saklarsak, iç tür, işaretçinin değerine bakılmaksızın* int olacaktır: (T = * int, V = nil). Bu nedenle, bu tür bir arayüz değeri, içindeki işaretçi değeri V sıfır olduğunda bile sıfır

olmayacaktır. Bu durum kafa karıştırıcı olabilir ve bir sıfır değeri gibi bir arayüz değerinin içinde saklandığında ortaya çıkar. hata dönüşü:

```
func returnsError() error {
    var p *MyError = nil
    if bad() {
        p = ErrBad
    }
    return p // her zaman nil olmayan hata döndürür
}
```

Her şey yolunda giderse, işlev bir nil p döndürür, dolayısıyla dönüş değeri bir hata arabirim değeridir (T = * MyError, V = nil). Bu, arayanın döndürdüğü hatayı nil ile karşılaştırması durumunda, kötü bir şey olmasa bile her zaman bir hata varmış gibi görüneceği anlamına gelir. Çağırana uygun bir sıfır hatası döndürmek için, işlevin açık bir nil döndürmesi gerekir:

```
func returnsError() error {
    if bad() {
        return ErrBad
    }
    return nil
}
```

Hataları döndüren fonksiyonların, hatanın doğru şekilde oluşturulmasını garantilemeye yardımcı olmak için *MyError gibi somut bir tür yerine (yukarıda yaptığımız gibi) imzalarında her zaman hata türünü kullanmaları iyi bir fikirdir. Örnek olarak, [os.Open](#), sıfır değilse bile, her zaman somut [*os.PathError](#) türünde olsa bile bir hata döndürür. Burada açıklananlara benzer durumlar, arayüzler her kullanıldığında ortaya çıkabilir. Arayüzde herhangi bir somut değer saklandıysa arayüzün nil olmayacağını unutmayın. Daha fazla bilgi için [Yansıma Yasaları](#)'na bakın.*

→ C'de olduğu gibi neden untagged unions yok?

Untagged unions, Go'nun bellek güvenliği garantilerini ihlal eder.

→ Go'da neden varyant türleri yok?

Cebirsel türler olarak da bilinen değişken türleri, bir değer diğer türlerden birini, ancak yalnızca bu türleri alabileceğini belirtmenin bir yolunu sağlar. Sistem programlamasında yaygın bir örnek, bir hatanın, örneğin bir ağ hatası, bir güvenlik hatası veya bir uygulama hatası olduğunu belirtir ve arayanın, hatanın türünü inceleyerek sorunun kaynağını ayırt etmesine izin verir.

Başka bir örnek, her bir düğümün farklı bir türde olabileceği bir sözdizimi ağacıdır: bildirim, ifade, atama vb. Go'ya varyant türleri eklemeyi düşündük, ancak tartışmadan sonra, arayüzlerle kafa karıştırıcı şekillerde çakiştıkları için bunları dışarıda bırakmaya karar verdik.

Bir varyant türünün öğelerinin kendileri arayüz olsaydı ne olurdu? Ayrıca, hangi varyant türlerinin adreslendiğinden bazıları zaten dil tarafından kapsanmaktadır. Hata örneğini, hatayı tutmak için bir arayüz değeri ve durumları ayırt etmek için bir tür anahtarı kullanarak ifade etmek kolaydır. Sözdizimi ağacı örneği, o kadar zarif olmasa da yapılabilir.

→ Go neden ortak değişken sonuç türlerine sahip değil?

Kovaryant (ortak değişken) sonuç türleri, benzer bir arayüzün

```
type Copyable interface {  
    Copy() interface{ }  
}
```

yöntemle karşılanmalı

func (v Value) Copy() Value

Çünkü Değer, boş arayüzü uygular. Go metodunda türlerin tam olarak eşleşmesi gerekir, bu nedenle Value Copyable'ı uygulamaz. Go, bir türün ne yaptığı kavramını - yöntemlerini - türün uygulamasından ayırır. İki yöntem farklı türler döndürürse, aynı şeyi yapmazlar. Kovaryant sonuç türleri isteyen programcılar genellikle arayüzler aracılığıyla bir tür hiyerarşisini ifade etmeye çalışırlar. Go'da arayüz ve uygulama arasında temiz bir ayırım olması daha doğaldır.

□ Değerler

→ Go neden örtük sayısal dönüşümler sağlamaz?

C'deki sayısal türler arasında otomatik dönüşümün rahatlığı, neden olduğu kafa karışıklığından daha ağır basmaktadır. Bir ifade ne zaman işaretlidir? Değer ne kadar büyük? Taşıyor mu? Sonuç, üzerinde yürütüldüğü makineden bağımsız olarak taşınabilir mi? Ayrıca derleyiciyi karmaşıklaştırır; "Olağan aritmetik dönüşümlerin" uygulanması kolay değildir ve mimariler arasında tutarsızdır.

Taşınabilirlik nedenlerinden dolayı, koddaki bazı açık dönüşümler pahasına işleri net ve anlaşılır hale getirmeye karar verdik. Go'daki sabitlerin tanımı - işaretli ve boyut ek açıklamaları içermeyen keyfi kesinlik değerleri - yine de sorunları önemli ölçüde iyileştirir. Bununla ilgili bir ayrıntı, C'den farklı olarak int ve int64'ün, int 64 bitlik bir tür olsa bile farklı türler olmasıdır. Int türü geneldir; Bir tamsayının kaç bit tuttuğunu önemsiyorsanız, Go sizi açık sözlü olmaya teşvik eder.

→ Sabitler Go'da nasıl çalışır?

Go, farklı sayısal türlerdeki değişkenler arasında dönüşüm konusunda katı olmasına rağmen, dildeki sabitler çok daha esnektir. 23, 3.14159 ve math.Pi gibi değişmez sabitler, keyfi bir hassasiyetle ve taşma veya yetersizlik olmadan bir tür ideal sayı alanı kaplar. Örneğin, math.Pi'nin değeri kaynak kodda 63 basamak olarak belirtilir ve değeri içeren sabit ifadeler, bir float64'ün tutabileceğinin ötesinde hassasiyeti korur. Yalnızca sabit veya sabit ifade bir değişkene (programdaki bir bellek konumuna) atandığında, normal kayan nokta özelliklerine ve hassasiyetine sahip bir “bilgisayar” numarası olur.

Ayrıca, bunlar tiplenmiş değerler değil, sadece sayılar oldukları için, Go'daki sabitler değişkenlere göre daha özgürce kullanılabilir, böylece katı dönüştürme kuralları etrafındaki bazı gariplikleri yumuşatır. Aşağıdaki gibi ifadeler yazılabilir:

```
sqrt2 := math.Sqrt(2)
```

derleyiciden şikayet etmeden ideal sayı 2, math.Sqrt çağrısı için güvenli ve doğru bir şekilde float64'e dönüştürülebilir.

[Sabitler](#) başlıklı bir blog yazısı bu konuyu daha ayrıntılı olarak araştırıyor. Ve bu dökümandan sabitleri inceleyebilirsiniz.

```
{% page-ref page="sabitler.md" %}
```

➔ Map neden built-in (yerleşik)?

Stringler ile aynı nedeni şudur: o kadar güçlü ve önemli bir veri yapısıdır ki sözdizimsel destek ile mükemmel bir uygulama sağlamak, programlamayı daha keyifli hale getirir. Go'nun map uygulamasının, kullanımların büyük çoğunluğuna hizmet edecek kadar güçlü olduğuna inanıyoruz. Belirli bir uygulama özel bir uygulamadan

yararlanabiliyorsa, bir tane yazmak mümkündür, ancak sözdizimsel olarak o kadar kullanışlı olmayacaktır; bu makul bir değiş tokuş gibi görünüyor.

→ Map neden dilimlere (slices) anahtar (key) olarak izin vermiyor?

Harita arama, dilimlerin uygulanmadığı bir eşitlik operatörü gerektirir. Eşitliği uygulamazlar çünkü eşitlik bu tür türlerde iyi tanımlanmamıştır; sık ve derin karşılaştırma, işaretçi ile değer karşılaştırması, özyinelemeli türlerle nasıl başa çıkılacağı gibi birçok husus vardır. Bu konuyu tekrar gözden geçirebiliriz - ve dilimler için eşitliği uygulamak mevcut programları geçersiz kılmaz - ancak dilimlerin eşitliğinin ne anlama geldiği konusunda net bir fikir olmadan, şimdilik bunu dışarıda bırakmak daha kolaydı.

Go 1'de, önceki sürümlerden farklı olarak, yapılar ve diziler için eşitlik tanımlanmıştır, bu nedenle bu tür türler eşleme anahtarları olarak kullanılabilir. Yine de dilimler hala bir eşitlik tanımına sahip değil.

→ Diziler değer iken neden map'ler, dilimler ve kanallar referanslarıdır?

Bu konuyla ilgili çok fazla tarih var. Eskiden, map'ler ve kanallar sözdizimsel olarak işaretçilerdi ve işaretçi olmayan bir örneği bildirmek veya kullanmak imkansızdı. Ayrıca dizilerin nasıl çalışması gerektiğiyle uğraştık. Sonunda, işaretçilerin ve değerlerin katı bir şekilde ayrılmasının, dili kullanmayı zorlaştırdığına karar verdik. Bu türlerin ilişkili, paylaşılan veri yapılarına referans görevi görecektik şekilde değiştirilmesi bu sorunları çözdü. Bu değişiklik, dile üzücü bir karmaşıklık kattı ancak kullanılabilirlik üzerinde büyük bir

etkiye sahipti: Go, tanıtıldığında daha üretken ve rahat bir dil haline geldi.

□ Kod Yazma

→ Kütüphaneler nasıl belgelenir?

Go'da yazılmış, kaynak koddan paket belgelerini çıkaran ve bunu bildirimlere, dosyalara vb. Bağlantılar içeren bir web sayfası olarak hizmet veren bir program var. golang.org/pkg/ adresinde bir örnek çalışıyor. Aslında, godoc sitenin tamamını golang.org/ adresinde uygulamaktadır.

Bir godoc örneği, görüntülediği programlarda sembollerin zengin, etkileşimli statik analizlerini sağlamak üzere yapılandırılabilir; detaylar [burada](#) listelenmiştir.

Komut satırından belgelere erişim için, [go](#) aracı aynı bilgilere bir metinsel arayüzü sağlayan bir [doc](#) alt komutunu vardır.

→ Go programlama stili kılavuzu var mı?

Kesinlikle tanınabilir bir “Go stili” olmasına rağmen açık bir stil kılavuzu yoktur.

Go, adlandırma, düzen ve dosya organizasyonu ile ilgili kararlara rehberlik edecek kurallar oluşturmuştur. [Effective Go](#) belgesi bu konularla ilgili bazı tavsiyeler içermektedir. Daha doğrusu, gofmt programı, amacı düzen kurallarını uygulamak olan güzel bir yazıcıdır; yorumlamaya izin veren olağan yapılması ve yapılmaması gerekenler özetinin yerini alır. Depodaki tüm Go kodu ve açık kaynak dünyasının büyük çoğunluğu gofmt aracılığıyla çalıştırılmıştır.

[Go Code Review Comments](#) başlıklı belge, programcılar tarafından genellikle gözden kaçırılan Go deyiminin

ayrıntlarıyla ilgili çok kısa denemelerden oluşan bir derlemedir. Go projeleri için kod incelemeleri yapan kişiler için kullanışlı bir referanstır.

→ Yamaları Go kütüphanelerine nasıl gönderirim?

Kütüphane kaynakları, arşivin src dizinindedir. Önemli bir değişiklik yapmak istiyorsanız, lütfen başlamadan önce posta listesinde tartışın.

Nasıl ilerleyeceğiniz hakkında daha fazla bilgi için [Go projesine katkıda bulunmak](#) belgesine bakın.

→ “go get” bir depoyu klonlarken neden HTTPS kullanıyor?

Şirketler genellikle giden trafiğe yalnızca standart TCP bağlantı noktaları 80 (HTTP) ve 443 (HTTPS) üzerinden izin vererek, TCP bağlantı noktası 9418 (git) ve TCP bağlantı noktası 22 (SSH) dahil olmak üzere diğer bağlantı noktalarında giden trafiği engeller. HTTP yerine HTTPS kullanırken, git varsayılan olarak sertifika doğrulamasını zorlayarak ortadaki adam, gizli dinleme ve kurcalama saldırılarına karşı koruma sağlar. Bu nedenle go getkomutu, güvenlik için HTTPS kullanır. Git, HTTPS üzerinden kimlik doğrulaması yapacak veya HTTPS yerine SSH kullanacak şekilde yapılandırılabilir. HTTPS üzerinden kimlik doğrulaması yapmak için \$ HOME / .netrc dosyasına git'in başvurduğu bir satır ekleyebilirsiniz:

```
machine github.com login USERNAME password APIKEY
```

GitHub hesapları için parola [kişisel bir erişim belirteci](#) olabilir.

Git, belirli bir önekle eşleşen URL'ler için HTTPS yerine SSH kullanacak şekilde de yapılandırılabilir. Örneğin, tüm GitHub erişiminde SSH kullanmak için şu satırları ~/.gitconfig dosyanıza ekleyin:

```
[url "ssh://git@github.com/"]  
  insteadOf = https://github.com/
```

→ “Go get” kullanarak paket sürümlerini nasıl yönetmeliyim?

Projenin başlangıcından bu yana, Go'nun açık bir paket sürümleri kavramı yoktu, ancak bu değişiyor. Versiyon oluşturma, özellikle büyük kod tabanlarında önemli bir karmaşıklık kaynağıdır ve tüm Go kullanıcılarına sağlanmaya uygun olacak kadar geniş bir çeşitlilikteki durumlarda ölçekte iyi çalışan bir yaklaşım geliştirmek biraz zaman almıştır. Go 1.11 sürümü, Go modülleri biçiminde go komutuna paket sürüm oluşturma için yeni, deneysel destek ekler. Daha fazla bilgi için [Go 1.11 sürüm notlarına](#) ve [go komutu belgelerine](#) bakın.

Gerçek paket yönetimi teknolojisinden bağımsız olarak, “go get” ve daha büyük Go araç zinciri, farklı içe aktarma yollarına sahip paketlerin izolasyonunu sağlar. Örneğin, standart kitaplığın html/template ve text/template, her ikisi de “package template” olsa bile bir arada bulunur.

Bu gözlem, paket yazarları ve paket kullanıcıları için bazı tavsiyelere yol açar. Kamusal kullanıma yönelik paketler, geriye dönük uyumluluğu korumaya çalışmalıdır. gelişmek. [Go 1 uyumluluk yönergeleri](#) burada iyi bir referanstır: dışa aktarılan adları kaldırmayın, etiketli bileşik değişmez değerleri teşvik edin, vb. Farklı işlevler gerekiyorsa, eskisini değiştirmek yerine yeni bir ad ekleyin. Tam bir ara verilmesi gerekiyorsa, yeni bir içe aktarma yolu ile yeni bir paket

oluşturun. Harici olarak sağlanan bir paket kullanıyorsanız ve beklenmedik şekillerde değişebileceğinden endişeleniyorsanız, ancak henüz Go modüllerini kullanmıyorsanız, en basit çözüm onu yerel deponuza kopyalamaktır. Bu, Google'ın dahili olarak kullandığı yaklaşımdır ve “vendoring” adı verilen bir teknik aracılığıyla go komutuyla desteklenir. Bu, bağımlılığın bir kopyasını, onu yerel bir kopya olarak tanımlayan yeni bir içe aktarma yolu altında depolamayı içerir. Ayrıntılar için [tasarım belgesi](#)ne bakın.

□ İşaretçiler ve Tahsis

→ Fonksiyon parametreleri ne zaman değere göre aktarılır?

C ailesindeki tüm dillerde olduğu gibi, Go'daki her şey değer bazında aktarılır. Diğer bir deyişle, bir işlev, değeri parametreye atayan bir atama ifadesi varmış gibi, her zaman iletilmekte olan şeyin bir kopyasını alır. Örneğin, bir fonksiyona bir int değerinin iletilmesi, int'in bir kopyasını oluşturur ve bir işaretçi değerinin iletilmesi, işaretçinin bir kopyasını oluşturur, ancak gösterdiği verilerin bir kopyasını oluşturmaz. (Bunun yöntem alıcılarını nasıl etkilediğiyle ilgili bir tartışma için sonraki bölüme bakın.) Map ve dilim (slice) değerleri, işaretçiler gibi davranır: bunlar, temel alınan map'e veya dilim verilerine işaretçiler içeren tanımlayıcılardır.

Bir map'in veya dilim değerinin kopyalanması, işaret ettiği verileri kopyalamaz. Bir arayüz değerinin kopyalanması, arayüz değerinde depolanan şeyin bir kopyasını oluşturur. Arayüz değeri bir yapı (struct) içeriyorsa, arayüz değerinin kopyalanması yapının bir kopyasını oluşturur. Arabirim değeri bir işaretçi tutuyorsa, arayüz değerinin kopyalanması

işaretçinin bir kopyasını oluşturur, ancak yine işaret ettiği veriyi oluşturmaz. Bu tartışmanın, işlemlerin anlambilim. Gerçek uygulamalar, optimizasyonlar anlambilimini değiştirmedeği sürece kopyalamayı önlemek için optimizasyonları uygulayabilir.

→ Bir arayüze işaretçi ne zaman kullanmalıyım?

Neredeyse hiç. Arayüz değerlerine yönelik işaretçiler yalnızca, gecikmeli değerlendirme için bir arayüz değerinin türünü gizlemeyi içeren nadir, zor durumlarda ortaya çıkar.

Bir arayüz bekleyen bir işleve bir arayüz değerine bir işaretçi iletmek yaygın bir hatadır. Derleyici bu hatadan şikayet eder, ancak durum yine de kafa karıştırıcı olabilir, çünkü bazen bir [arayüzü karşılamak için bir işaretçi](#) gerekir. Buradaki fikir, somut bir tipe bir işaretçi bir arayüzü karşılayabilmesine rağmen, bir istisna dışında, bir arayüze yönelik bir işaretçinin bir arayüze asla karşılamayacağıdır.

Değişken tanımlamayı düşünün,

```
var w io.Writer
```

fmt.Fprintf yazdırma fonksiyonu, ilk argüman olarak io.Writer'ı karşılayan bir değer alır - bu, kurallı Write metodunu uygulayan bir şeydir. Böylece yazabiliriz

Ancak w'nin adresini geçerse, program derlenmeyecektir.

```
fmt.Fprintf(&w, "hello, world\n") //Derleme-zamanı hatası.
```

Bunun tek istisnası, herhangi bir değer, hatta bir arayüze işaretçi bile, boş arayüze türündeki bir değişkene (interface{}) atanabilmesidir. Öyle olsa bile, değer bir arayüze işaretçi olması neredeyse kesinlikle bir hatadır; sonuç kafa karıştırıcı olabilir.

→ Metodları (struct fonksiyonlar) değerler veya işaretçiler üzerinde tanımlamalı mıyım?

```
func (s *MyStruct) pointerMethod() { } // işaretçi metod  
func (s MyStruct) valueMethod() { } // değer metod
```

İşaretçilere alışkın olmayan programcılar için bu iki örnek arasındaki ayrım kafa karıştırıcı olabilir, ancak durum aslında çok basittir. Bir tür üzerinde bir metodu tanımlarken, alıcı (yukarıdaki örneklerde s) tam olarak metodun bir argümanıymış gibi davranır. Alıcının bir değer olarak mı yoksa bir işaretçi olarak mı tanımlanacağı aynı sorudur, o halde, bir fonksiyon bağımsız değişkeninin bir değer mi yoksa bir işaretçi mi olması gerektiği sorusudur.

Dikkat edilmesi gereken birkaç nokta var. İlk ve en önemlisi, metodun alıcıyı değiştirmesi gerekiyor mu? Eğer öyleyse, alıcının bir işaretçi olması gerekir. (Dilimler ve map'ler referans görevi görür, bu nedenle hikayeleri biraz daha inceliklidir, ancak örneğin bir yöntemdeki bir dilimin uzunluğunu değiştirmek için alıcının yine de bir işaretçi olması gerekir.)

Yukarıdaki örneklerde, pointerMethod alanlarını değiştirirse s, arayan bu değişiklikleri görecektir, ancak valueMethod, arayanın argümanının bir kopyasıyla çağrılır (bu, bir değeri iletmenin tanımıdır), bu nedenle yaptığı değişiklikler görünmez olacaktır arayana.

Bu arada, Java metodlarında alıcılar her zaman işaretçilerdir, ancak işaretçi doğaları bir şekilde gizlenmiştir (ve dile değer alıcıları eklemek için bir öneri vardır). Sıradışı olan, Go'daki değer alıcılarıdır. İkincisi, verimliliğin dikkate alınmasıdır. Alıcı büyükse, örneğin büyük bir yapıya sahipse, bir işaretçi alıcı kullanmak çok daha ucuz olacaktır.

Sırada tutarlılık var. Türün bazı metodlarının işaretçi alıcıları olması gerekiyorsa, geri kalanı da olmalıdır, bu nedenle metod kümesi türün nasıl kullanıldığına bakılmaksızın tutarlıdır. Ayrıntılar için [metod kümeleri](#) bölümüne bakın.

Temel türler, dilimler ve küçük yapılar gibi türler için, bir değer alıcısı çok ucuzdur, bu nedenle metodun anlambilimi bir işaretçi gerektirmedikçe, bir değer alıcısı verimli ve nettir.

→ Make ve New arasındaki fark nedir?

Kısaca: new, bellek ayırır. Make ise dilim, map ve kanal türlerini başlatır.

Daha fazla ayrıntı için [Effective Go'nun ilgili bölümü](#)ne bakın.

→ Bir int türün 64bitlik makinedeki büyüklüğü nedir?

Int ve uint boyutları uygulamaya özgüdür ancak belirli bir platformda birbirleriyle aynıdır. Taşınabilirlik için, belirli bir değer boyutuna dayanan kod, int64 gibi açıkça boyutlandırılmış bir tür kullanılmalıdır. 32-bit makinelerde derleyiciler varsayılan olarak 32-bit tamsayılar kullanırken, 64-bit makinelerde tamsayılar 64 bit'e sahiptir. (Tarihsel olarak bu her zaman doğru değildi.)

Öte yandan, kayan nokta skalerleri ve karmaşık türler her zaman boyutlandırılır (kayan nokta veya karmaşık temel türler yoktur), çünkü programcılar kayan noktalı (burada float'dan bahsediyor) sayıları kullanırken hassasiyetin farkında olmalıdır. Bir (türlenmemiş - untyped) kayan nokta sabiti için kullanılan varsayılan tür float64'tür. Böylece foo: = 3.0, float64 türünde bir değişken foo bildirir. Bir

(türlenmemiş) sabit tarafından başlatılan `float32` değişkeni için değişken türü, değişken bildiriminde açıkça belirtilmelidir:

```
var foo float32 = 3.0
```

Alternatif olarak, sabite `foo: = float32 (3.0)` 'da olduğu gibi dönüşümlü bir tür verilmelidir.

→ Bir değişkenin yığın üzerinde mi yoksa yığın üzerinde tahsis edildiğini nasıl anlarım?

Doğruluk açısından, bilmenize gerek yok. Go'daki her değişken, kendisine referans olduğu sürece var olur. Uygulama tarafından seçilen depolama konumu, dilin anlambilimiyle ilgisizdir.

Depolama konumu, verimli programlar yazma üzerinde bir etkiye sahiptir. Mümkün olduğunda, Go derleyicileri, o işlevin yığın çerçevesindeki bir işleve yerel olan değişkenleri tahsis eder. Ancak, derleyici işlev döndükten sonra değişkene başvurulmadığını kanıtlayamazsa, derleyicinin işaretçi hatalarının sarkmasını önlemek için değişkeni çöpte toplanan yığın üzerinde tahsis etmesi gerekir.

Ayrıca, yerel bir değişken çok büyükse, onu yığın yerine yığın üzerinde depolamak daha mantıklı olabilir. Mevcut derleyicilerde, bir değişkenin adresi alınmışsa, bu değişken öbek üzerinde tahsisat için bir adaydır. Bununla birlikte, temel bir kaçış analizi, bu tür değişkenlerin işlevin geri dönüşünü geçemez ve tanımsız yığındır.

→ Go işlemim neden bu kadar çok sanal bellek kullanıyor?

Go bellek ayırıcısı, ayırmalar için bir alan olarak büyük bir sanal bellek bölgesini ayırır. Bu sanal bellek, belirli Go işlemi için yereldir; rezervasyon, diğer bellek işlemlerini mahrum etmez.

Bir Go işlemine ayrılan gerçek bellek miktarını bulmak için Unix `top` komutunu kullanın ve RES (Linux) veya RSIZE (macOS) sütunlarına bakın.

□ Eşzamanlılık

→ Hangi işlemler atomiktir? Mutexler ne olacak?

Go'daki işlemlerin atomikliğine ilişkin bir açıklama [Go Bellek Modeli](#) belgesinde bulunabilir.

Düşük seviyeli senkronizasyon ve atomik ilkeller, [sync](#) ve [sync/atomic](#) paketlerinde mevcuttur.

Bu paketler, referans sayılarını artırmak veya küçük ölçekli karşılıklı dışlamayı garanti etmek gibi basit görevler için iyidir. Eşzamanlı sunucular arasında koordinasyon gibi daha yüksek seviyeli işlemler için, daha yüksek seviyeli teknikler daha güzel programlara yol açabilir ve Go, bu yaklaşımı kendi düzenleri ve kanalları aracılığıyla destekler. Örneğin, programınızı belirli bir veri parçasından her seferinde yalnızca bir gorutin sorumlu olacak şekilde yapılandırabilirsiniz. Bu yaklaşım, orijinal [Go atasözü](#) ile özetlenmiştir.

Hafızayı paylaşarak iletişim kurmayın. Bunun yerine, iletişim kurarak hafızayı paylaşın. Bu kavramın ayrıntılı bir tartışması için, [İletişimi Kurarak Belleği Paylaşma kod yürüyüşüne ve ilgili](#) makalesine bakın. Büyük eşzamanlı programlar muhtemelen bu iki araç setinden ödünç alır.

→ Programım neden daha fazla CPU ile daha hızlı çalışmıyor?

Bir programın daha fazla CPU ile daha hızlı çalışıp çalışmadığı, çözmekte olduğu soruna bağlıdır. Go dili, gorutinler ve kanallar gibi eşzamanlılık ilkeleri sağlar, ancak eşzamanlılık yalnızca temeldeki sorun doğası gereği paralel olduğunda paralellik sağlar.

Doğası gereği sıralı olan sorunlar, daha fazla CPU eklenerek hızlandırılmazken, paralel olarak yürütülebilen parçalara ayrılabilen sorunlar bazen dramatik bir şekilde hızlandırılabilir. Bazen daha fazla CPU eklemek bir programı yavaşlatabilir. Pratik anlamda, eşitleme veya iletişim için yararlı hesaplamalar yapmaktan daha fazla zaman harcayan programlar, birden çok işletim sistemi iş parçacığı kullanırken performans düşüşü yaşayabilir. Bunun nedeni, iş parçacıkları arasında veri geçişinin, önemli bir maliyeti olan bağlamları değiştirmeyi gerektirmesidir ve bu maliyet daha fazla CPU ile artabilir. Örneğin, Go spesifikasyonundaki ana elek örneğinin pek çok gorutini başlatmasına rağmen önemli bir paralelliği yoktur; sayısını artırmak thread'ler (CPU'lar) hızlandırmaktan çok yavaşlatır.

Bu konu hakkında daha fazla ayrıntı için [Concurrency is not Parallelism](#) başlıklı konuşmaya bakın.

→ CPU sayısını nasıl kontrol edebilirim?

Yürütülen programlı dizinler için aynı anda kullanılabilen CPU sayısı, varsayılan değeri mevcut CPU çekirdeği sayısı olan GOMAXPROCS kabuk ortam değişkeni tarafından kontrol edilir. Paralel yürütme potansiyeli olan programlar bu nedenle bunu varsayılan olarak çoklu CPU'lu bir makinede gerçekleştirmelidir. Kullanılacak paralel CPU sayısını değiştirmek için ortam değişkenini ayarlayın veya çalışma

zamanı desteğini farklı sayıda iş parçacığı kullanacak şekilde yapılandırmak için çalışma zamanı paketinin benzer adlandırılmış işlevini kullanın. 1 olarak ayarlamak, bağımsız gorutinleri sırayla yürütmeye zorlayarak gerçek paralellik olasılığını ortadan kaldırır. Çalışma zamanı, birden çok bekleyen G / Ç isteğine hizmet vermek için GOMAXPROCS değerinden daha fazla iş parçacığı ayırabilir.

GOMAXPROCS yalnızca aynı anda kaç tane gorutin çalıştırabileceğini etkiler; sistem çağrılarında keyfi olarak daha fazlası engellenebilir. Go'nun gorutin programlayıcısı, zamanla gelişmesine rağmen olması gerektiği kadar iyi değildir. Gelecekte daha iyi olabilir OS iş parçacığı kullanımını optimize eder. Şimdilik, performans sorunları varsa, GOMAXPROCS'u uygulama bazında ayarlamak yardımcı olabilir.

→ Gorutin olarak çalışan kapanışlarda ne olur?

Eşzamanlı kapanışlar kullanıldığında bazı karışıklıklar ortaya çıkabilir. Aşağıdaki programı düşünün:

```
func main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := range values {
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }

    // çıkmadan önce tüm goroutinlerin tamamlanmasını bekle
    for _ = range values {
        <-done
    }
}
```

Bir kimse yanlışlıkla çıktı olarak a, b, c'yi görmeyi bekleyebilir. Bunun yerine muhtemelen göreceğiniz şey c, c, c'dir. Bunun nedeni, döngünün her yinelemesinin v değişkeninin aynı örneğini kullanmasıdır, bu nedenle her kapanış bu tek değişkeni paylaşır. Kapatma çalıştığında, `fmt.Println` yürütüldüğünde v değerini yazdırır, ancak gorutin başlatıldıktan sonra v değiştirilmiş olabilir. Bunu ve diğer sorunları ortaya çıkmadan önce tespit etmeye yardımcı olmak için [go vet](#) komutunu kullanın.

v'nin geçerli değerini, başlatıldığında her kapanışa bağlamak için, her yinelemede yeni bir değişken oluşturmak için iç döngü değiştirilmelidir. Bir yol, değişkeni kapanışa bir argüman olarak iletmektir:

```
for _, v := range values {
    go func(u string) {
        fmt.Println(u)
        done <- true
    }(v)
}
```

Bu örnekte, v'nin değeri anonim işleve argüman olarak aktarılır. Bu değere daha sonra fonksiyonun içinde u değişkeni olarak erişilebilir. Daha da kolay olanı, tuhaf görünebilecek ancak Go'da iyi çalışan bir tanımlama stili kullanarak yeni bir değişken oluşturmaktır:

```
for _, v := range values {
    v := v // create a new 'v'.
    go func() {
        fmt.Println(v)
        done <- true
    }()
}
```

Her yineleme için yeni bir değişken tanımlamayan dilin bu davranışı, geçmişte bakıldığında bir hata olabilir. Daha

sonraki bir sürümde ele alınabilir, ancak uyumluluk için Go sürüm 1’de değiştirilemez.

□ Fonksiyonlar ve Metodlar (Structlar için fonksiyonlar)

→ T ve * T’nin neden farklı metod atamaları var?

[Go spesifikasyonunun](#) dediği gibi, T türünün metod kümesi, alıcı türü T olan tüm metodlardan oluşurken, karşılık gelen işaretçi türü *T*, *alıcı T* veya **T* olan tüm yöntemlerden oluşur. Bu, **T yöntem kümesidir. T’yi içerir, ancak tersini içermez. Bu ayrım, bir arabirim değeri bir T işaretçisi içeriyorsa, bir yöntem çağrısı işaretçinin başvurusunu kaldırarak bir değer elde edebildiği için ortaya çıkar, ancak bir arabirim değeri bir T değeri içeriyorsa, bir işaretçi elde etmek için bir yöntem çağrısı için güvenli bir yol yoktur. (Bunu yapmak, bir yöntemin arayüz içindeki değerin içeriğini değiştirmesine izin verir ve bu, dil spesifikasyonu tarafından izin verilmemektedir.)* Derleyicinin yönteme geçmek için bir değerin adresini alabildiği durumlarda bile, yöntem değeri değiştirirse, değişiklikler çağırıcıda kaybolur. Örnek olarak, `bytes.Buffer`’ın `Write` yöntemi bir işaretçi yerine bir değer alıcısı kullanıyorsa, bu kod:

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

standart girdiyi `buf`’ın kendisine değil `buf`’ın bir `Copy` elemanına kopyalar. Bu neredeyse hiçbir zaman istenen davranış değildir.

□ Kontrol Akışı

→ Go neden ?: operatörüne sahip değil?

Go'da üçlü test işlemi yoktur. Aynı sonucu elde etmek için aşağıdakileri kullanabilirsiniz:

```
if expr {  
    n = trueVal  
} else {  
    n = falseVal  
}
```

Go'da olmayan ?:, dilin tasarımcılarının işlemin aşılmaz karmaşık ifadeler yaratmak için çok sık kullanıldığını görmeleridir. If-else formu, daha uzun olmasına rağmen, tartışmasız bir şekilde daha nettir. Bir dilin yalnızca bir koşullu kontrol akışı yapısına ihtiyacı vardır.

□ Paketler ve Testing

→ Nasıl çok dosyalı paket oluştururum?

Paketin tüm kaynak dosyalarını tek başlarına bir dizine koyun. Kaynak dosyalar isteğe bağlı olarak farklı dosyalardan öğelere başvurabilir; ileri tanımlamalara veya bir başlık dosyasına gerek yoktur. Birden çok dosyaya bölünmek dışında, paket tek dosyalık bir paket gibi derlenecek ve test edilecektir.

→ Nasıl birim testi yazarım?

Paket kaynaklarınızla aynı dizinde `_test.go` ile biten yeni bir dosya oluşturun. Bu dosyanın içine, "testing" i içe aktarın ve formun işlevlerini yazın

```
func TestFoo(t *testing.T) {  
    ...  
}
```

Bu dizindeyken `go test` komutunu çalıştırın. Bu komut dosyası Test fonksiyonlarını bulur, bir test ikili dosyası oluşturur ve çalıştırır.

Daha fazla ayrıntı için [How to Write Go Code](#) belgesine, [testing](#) paketine ve [go test](#) alt komutuna bakın.

→ Test için en sevdiğim yardımcı fonksiyon nerede?

Go'nun standart test paketi, birim testleri yazmayı kolaylaştırır, ancak onaylama işlevleri gibi diğer dillerin test çerçevelerinde sağlanan özelliklerden yoksundur. Bu belgenin daha önceki bir bölümü Go'nun neden iddialara sahip olmadığını ve aynı argümanlar testlerde `assert` kullanımı için de geçerli olduğunu açıkladı. Doğru hata işleme, biri başarısız olduktan sonra diğer testlerin çalıştırılmasına izin vermek anlamına gelir, böylece hatayı ayıklayan kişi neyin yanlış olduğunu tam olarak görebilir. `isPrime`'ın 2, 3, 5 ve 7 (veya 2, 4, 8 ve 16 için) için yanlış cevap verdiğini rapor etmek, `isPrime`'ın 2 için yanlış cevap verdiğini ve bu nedenle hayır cevabını vermekten daha yararlıdır. daha fazla test yapıldı. Test hatasını tetikleyen programcı, başarısız olan koda aşına olmayabilir.

İyi bir hata mesajı yazmak için harcanan zaman artık daha sonra test sona erdiğinde karşılığını veriyor. Bununla ilgili bir nokta, test çerçevelerinin, koşullu ifadeler, kontroller ve baskı mekanizmaları ile kendi mini dillerine dönüşme eğiliminde olmasıdır. ancak Go tüm bu yeteneklere zaten sahiptir; neden onları yeniden yarattın? Go'da testler yazmayı tercih ederiz; öğrenilmesi gereken daha az dildir ve bu yaklaşım, testleri basit ve anlaşılması kolay tutar.

İyi hatalar yazmak için gereken ekstra kod miktarı tekrarlayıcı ve ezici görünüyorsa, test, bir veri yapısında

tanımlanmış bir girdi ve çıktı listesi üzerinde yinelenen tabloya dayalıysa daha iyi çalışabilir (Go, veri yapısı değişmezleri için mükemmel bir desteğe sahiptir). İyi bir test ve iyi hata mesajları yazma işi daha sonra birçok test senaryosuna göre amortismanına tabi tutulacaktır. Standart Go kitaplığı, fmt paketi için [biçimlendirme testleri gibi açıklayıcı örnekler](#)le doludur.

→ X neden standart kütüphanede yok?

Standart kütüphanenin amacı, çalışma zamanını desteklemek, işletim sistemine bağlanmak ve biçimlendirilmiş G / Ç ve ağ iletişimi gibi birçok Go programının gerektirdiği temel işlevleri sağlamaktır. Ayrıca, kriptografi ve HTTP, JSON ve XML gibi standartlar için destek dahil olmak üzere web programlama için önemli öğeler içerir.

Neyin dahil edileceğini tanımlayan net bir kriter yok çünkü uzun zamandır bu tek Go kütüphanesi idi. Bununla birlikte, bugün neyin ekleneceğini tanımlayan kriterler var. Standart kitaplığa yeni eklemeler nadirdir ve dahil etme çıtası yüksektir. Standart kitaplığa dahil edilen kod, büyük bir sürekli bakım maliyeti taşır (genellikle orijinal yazar dışındaki kişiler tarafından karşılanır), [Go 1 uyumluluk taahhüdü](#)ne (API'deki herhangi bir kusur için engelleme düzeltmeleri) tabidir ve Go sürümüne tabidir.

Program, hata düzeltmelerinin kullanıcılara hızlı bir şekilde sunulmasını önler. Yeni kodların çoğu standart kitaplığın dışında yaşamalı ve [go tool](#) ile erişilebilir olmalıdır. git komuta al. Bu tür kodların kendi bakımcıları, yayın döngüsü ve uyumluluk garantileri olabilir. Kullanıcılar [godoc.org](#) adresinde paketleri bulabilir ve belgelerini okuyabilir. Standart kütüphanede log/syslog gibi gerçekten ait olmayan parçalar olsa da, Go 1 uyumluluk vaadi nedeniyle

kitaplıktaki her şeyi korumaya devam ediyoruz. Ancak çoğu yeni kodu başka bir yerde yaşamaya teşvik ediyoruz.

□ İmplementasyonlar

→ Derleyicileri oluşturmak için hangi derleyici teknolojisi kullanılır?

Go için birkaç üretim derleyicisi ve çeşitli platformlar için geliştirilmekte olan birkaç başka derleyici vardır. Varsayılan derleyici gc, go komutunun desteğinin bir parçası olarak Go dağıtımına dahildir. Gc, başlangıçta önyüklemenin zorlukları nedeniyle C dilinde yazılmıştır - bir Go ortamı kurmak için bir Go derleyicisine ihtiyacınız vardır. Ancak işler gelişti ve Go 1.5 sürümünden bu yana derleyici bir Go programı oldu. Derleyici, bu [tasarım belgesi](#)nde ve konuşmada açıklandığı gibi otomatik çeviri araçları kullanılarak C'den Go'ya dönüştürüldü.

Bu nedenle, derleyici artık “kendi kendine barındırılıyor”, bu da önyükleme sorunuyla yüzleşmemiz gerektiği anlamına geliyor. Çözüm, çalışan bir C kurulumunda olduğu gibi, çalışan bir Go kurulumuna sahip olmaktır. Kaynağından yeni bir Go ortamının nasıl ortaya çıkarılacağına hikayesi [burada](#) ve [burada](#) anlatılmaktadır.

Gc, Go'da özyinelemeli bir iniş ayrıştırıcıyla yazılır ve yine Go'da yazılan ancak Plan 9 yükleyiciye dayanan özel bir yükleyici kullanır, ELF / Mach-O / PE ikili dosyaları oluşturmak için. Projenin başlangıcında gc için LLVM kullanmayı düşündük, ancak performans hedeflerimizi karşılamak için çok büyük ve yavaş olduğuna karar verdik. Geriye dönüp bakıldığında daha önemli olan, LLVM ile başlamak, Go'nun gerektirdiği ancak standart C kurulumunun bir parçası olmayan bazı ABI ve yığın yönetimi

gibi ilgili deęişiklikleri uygulamaya koymayı zorlaştırırdı. Ancak şimdi yeni bir [LLVM uygulaması](#) bir araya gelmeye başlıyor.

Gccgo derleyicisi, standart GCC arka ucuna baęlı özyinelemeli bir iniş ayrıştırıcısı ile C ++ ile yazılmış bir ön uçtur. Go'nun bir Go derleyicisini uygulamak için iyi bir dil olduęu ortaya çıktı, ancak asıl amacı bu deęildi. Başından beri kendi kendine barındırılmaması, Go'nun tasarımının aęa baęlı sunucular olan orijinal kullanım durumuna odaklanmasına izin verdi. Go'nun kendisini erken derlemesine karar vermiş olsaydık, derleyici yapımı için daha çok hedeflenmiş bir dil bulabilirdik, bu deęerli bir hedef ama sahip olduęumuz deęil başlangıçta.

Gc bunları kullanmasa da (henüz), Go paketinde yerel bir sözcük ve ayrıştırıcı mevcuttur ve ayrıca yerel bir [tür denetleyici](#) de vardır.

→ Çalışma zamanı desteęi nasıl uygulanır?

Yine, önyükleme sorunları nedeniyle, çalışma zamanı kodu başlangıçta çoęunlukla C ile yazılmıştır (küçük bir montajlayıcı ile), ancak o zamandan beri Go'ya çevrilmiştir (bazı assembler bitleri hariç). Gccgo'nun çalışma zamanı desteęi glibc kullanır. Gccgo derleyicisi, altın baęlayıcıda yapılan son deęişikliklerle desteklenen, segmentli yığınlar adı verilen bir teknik kullanarak gorutinleri uygular. Go11vm benzer şekilde ilgili LLVM altyapısı üzerine inşa edilmiştir.

→ Basit programım neden büyük dosya boyutlu?

Gc araç zincirindeki baęlayıcı, varsayılan olarak statik baęlantılı ikili dosyalar oluşturur. Bu nedenle tüm Go ikili dosyaları, dinamik tür kontrollerini, yansımayı ve hatta panik

zamanı yığın izlerini desteklemek için gerekli çalışma zamanı tür bilgileriyle birlikte Go çalışma zamanını içerir.

Linux'ta gcc kullanılarak statik olarak derlenen ve bağlanan basit bir C “merhaba, dünya” programı, bir printf uygulaması da dahil olmak üzere yaklaşık 750 kB'dir. fmt.Printf kullanan eşdeğer bir Go programı birkaç megabayt ağırlığındadır, ancak bu daha güçlü çalışma zamanı desteği ve tür ve hata ayıklama bilgileri içerir. Gc ile derlenen bir Go programı -ldflags = -w bayrağına bağlanarak DWARF oluşturmayı devre dışı bırakarak ikili dosyadan hata ayıklama bilgilerini kaldırabilir, ancak başka bir işlev kaybı olmaz. Bu, ikili boyutu önemli ölçüde azaltabilir.

→ Kullanılmayan değişken/içe aktarım ile ilgili uyarıları durdurabilir miyim?

Kullanılmayan bir değişkenin varlığı bir hatayı gösterebilirken, kullanılmayan içe aktarmalar derlemeyi yavaşlatır, bir program zamanla kod ve programcılar biriktirdikçe önemli hale gelebilir. Bu nedenlerle Go, kullanılmayan değişkenler veya içe aktarmalar içeren programları derlemeyi reddeder, uzun vadeli oluşturma hızı ve program netliği için kısa vadeli kolaylık ticaretini yapar.

Yine de, kod geliştirirken, bu durumları geçici olarak oluşturmak yaygındır ve program derlenmeden önce bunları düzenlemenin gerekmesi can sıkıcı olabilir. Bazıları, bu kontrolleri kapatmak veya en azından uyarılara indirgemek için bir derleyici seçeneği talep etti. Böyle bir seçenek eklenmemiştir, çünkü derleyici seçenekleri dilin anlamını etkilememelidir ve Go derleyicisi uyarıları değil, yalnızca derlemeyi engelleyen hataları bildirdiği için. Hiçbir uyarı almamanın iki nedeni vardır. İlk olarak, şikayet etmeye değerse, kodu düzeltmeye değer. (Ve düzeltmeye değmiyorsa, buna değmez İkinci olarak, derleyicinin uyarılar

oluşturması, uygulamayı, derlemeyi gürültülü hale getirebilecek zayıf durumlar hakkında uyarmaya ve düzeltilmesi gereken gerçek hataları maskeleye teşvik eder. Yine de durumu ele almak kolaydır. Geliştirme sırasında kullanılmayan şeylerin devam etmesine izin vermek için boş tanımlayıcıyı kullanın.

```
import "unused"

// This declaration marks the import as used by referencing an
// item from the package.
var _ = unused.Item // TODO: Delete before committing!

func main() {
    debugData := debug.Profile()
    _ = debugData // Used only during debugging.
    ....
}
```

Günümüzde çoğu Go programcısı, [Goimports](#) adlı bir araç kullanıyor, bu da Go kaynak dosyasını doğru içe aktarmaları elde etmek için otomatik olarak yeniden yazıyor ve uygulamada kullanılmayan içe aktarma sorununu ortadan kaldırıyor. Bu program, bir Go kaynak dosyası yazıldığında otomatik olarak çalışması için çoğu düzenleyiciye kolayca bağlanır.

→ Virüs tarama yazılımım neden Go dağıtıma veya derlenmiş ikili dosyama virüs bulaştığını düşünüyor?

Bu, özellikle Windows makinelerde yaygın bir durumdur ve neredeyse her zaman yanlış bir pozitifdir. Ticari virüs tarama programları, diğer dillerden derlenenler kadar sık görmedikleri Go ikili dosyalarının yapısı nedeniyle genellikle karıştırılır. Go dağıtımını yeni yüklediyseniz ve sistem virüs bulaştığını bildirirse, bu kesinlikle bir hatadır.

Gerçekten kapsamlı olmak için, sağlama toplamını [indirilenler sayfasındaki](#)lerle karşılaştırarak indirmeyi doğrulayabilirsiniz. Her durumda, raporun hatalı olduğuna inanıyorsanız, lütfen virüs tarayıcınızın tedarikçisine bir hata bildirin. Belki zamanla virüs tarayıcıları Go programlarını anlamayı öğrenebilir.

□ Performans

→ Go, X karşılaştırmasında neden kötü performans gösteriyor?

Go'nun tasarım hedeflerinden biri, karşılaştırılabilir programlar için C'nin performansına yaklaştırmaktır, ancak bazı kıyaslamalarda, golang.org/x/exp/shootout'taki birkaçı da dahil olmak üzere, oldukça zayıf bir şekilde işliyor. En yavaş olanlar, karşılaştırılabilir performansın sürümlerinin Go'da bulunmadığı kütüphanelere bağlıdır. Örneğin, pidigits.go, çok duyarlıklı bir matematik paketine bağlıdır ve Go'nun aksine C sürümleri [GMP](#)'yi (optimize edilmiş derleyicide yazılmıştır) kullanır. Normal ifadeler (örneğin regex-dna.go) dayanan karşılaştırmalar, esasen Go'nun yerel [regexp paketi](#)ni PCRE gibi olgun, yüksek düzeyde optimize edilmiş düzenli ifade kitaplıklarıyla karşılaştırıyor.

Kıyaslama oyunları kapsamlı ayarlamalarla kazanılır ve kıyaslamaların çoğunun Go sürümleri dikkat gerektirir. Karşılaştırılabilir C ve Go programlarını ölçerseniz ([reverse-complement.go](#) bir örnektir), iki dilin ham performansta bu paketin gösterdiğinden çok daha yakın olduğunu göreceksiniz.

Yine de, iyileştirme için yer var. Derleyiciler iyidir ancak daha iyi olabilir, birçok kütüphane büyük performans çalışmasına ihtiyaç duyar ve çöp toplayıcı henüz yeterince

hızlı değildir. (Öyle olsa bile, gereksiz çöp üretmemeye özen göstermenin çok büyük bir etkisi olabilir.)

Her durumda, Go genellikle çok rekabetçi olabilir. Dil ve araçlar geliştikçe birçok programın performansında önemli gelişmeler olmuştur. Bilgilendirici bir örnek için [Go programlarını profileme](#) hakkındaki blog gönderisine bakın.

□ C'den Değişiklikler

→ Sözdizimi C'den neden bu kadar farklı?

Tanımlama sözdizimi dışında, farklılıklar büyük değildir ve iki arzudan kaynaklanır. İlk olarak, sözdizimi çok fazla zorunlu anahtar sözcük, tekrar veya gizem olmadan hafif hissetmelidir. İkinci olarak, dil analiz edilmesi kolay olacak şekilde tasarlanmıştır ve bir sembol tablosu olmadan ayrıştırılabilir. Bu, hata ayıklayıcılar, bağımlılık çözümleyicileri, otomatikleştirilmiş belge çıkarıcıları, IDE eklentileri vb. Gibi araçlar oluşturmayı çok daha kolaylaştırır. C ve onun soyundan gelenler bu bakımdan herkesin bildiği gibi zordur.

→ Tanımlamalar neden ters yönlü yapılır?

C'ye alışkınsanız, yalnızca geriye doğrudurlar. C'de, fikir bir değişkenin türünü ifade eden bir ifade gibi bildirilmesidir, bu güzel bir fikirdir, ancak tür ve ifade gramerleri çok iyi karışmaz ve sonuçlar kafa karıştırıcı olabilir; fonksiyon işaretlerini düşünün. Go, çoğunlukla ifade ve tür sözdizimini ayırır ve bu da işleri basitleştirir (işaretçiler için * öneki kullanmak, kuralı kanıtlayan bir istisnadır). C'de tanımlamalar

```
int* a, b;
```

Go'da ise

```
var a, b *int
```

her ikisinin de işaretçi olduğunu beyan eder. Bu daha net ve daha düzenli. Ayrıca: = kısa tanımlama formu, tam değişken bildiriminin şu sırayı sunması gerektiğini savunur : = değer

```
var a uint64 = 1
```

aşağıdaki ile aynı etkidir.

```
a := uint64(1)
```

Ayrıştırma, yalnızca ifade dilbilgisi olmayan türler için de farklı bir dilbilgisine sahip olarak basitleştirilmiştir; func ve chan gibi anahtar kelimeler her şeyi netleştirir.

Daha fazla ayrıntı için [Go'nun Tanımlama Sözdizimi](#) hakkındaki makaleye bakın.

→ Neden işaretçi aritmetiği yok?

Emniyet. İşaretçi aritmetiği olmadan, hatalı bir şekilde başarılı olan geçersiz bir adresi asla türetemeyecek bir dil oluşturmak mümkündür. Derleyici ve donanım teknolojisi, dizi indekslerini kullanan bir döngünün işaretçi aritmetiği kullanan bir döngü kadar verimli olabileceği noktaya kadar gelişmiştir. Ayrıca, işaretçi aritmetiğinin olmaması, çöp toplayıcının uygulanmasını basitleştirebilir.

→ Neden ++ ve - tanımlama değilde ifadedir? Ve neden ön ek değilde son ektir?

İşaretçi aritmetiği olmadan, ön ve son ek artırma operatörlerinin uygunluk değeri düşer. Bunları ifade hiyerarşisinden tamamen kaldırarak, ifade sözdizimi

basitleştirilir ve ++ ve - (f (i ++)) ve p [i] = q [++ i] 'yi düşünün) değerlendirme sırasına ilişkin karışık sorunlar da ortadan kaldırılır. . Sadeleştirme önemlidir. sonek ve öneke gelince, her ikisi de iyi çalışır, ancak son ek sürümü daha gelenekseldir; Önek ısrarı, adı ironik bir şekilde bir sonek artışını içeren bir dil kitaplığı olan STL ile ortaya çıktı.

→ Neden parantez var ama noktalı virgül yok? Ve neden sonraki satırda açılış ayracı koyamazsınız?

Go, C ailesindeki herhangi bir dille çalışan programcılarının aşına olduğu bir sözdizimi olan ifade gruplaması için ayrıç parantezleri kullanır. Ancak noktalı virgüller insanlar için değil, ayrıştırıcılar içindir ve biz onları olabildiğince ortadan kaldırmak istedik. Go, bu amaca ulaşmak için BCPL'den bir numara ödünç alır: İfadeleri ayıran noktalı virgüller biçimsel dilbilgisi içindedir, ancak bir ifadenin sonu olabilecek herhangi bir satırın sonuna sözcük yazar tarafından önden bakmadan otomatik olarak enjekte edilir.

Bu, pratikte çok iyi çalışır, ancak bir küme ayracı stilini zorlama etkisine sahiptir. Örneğin, bir işlevin açılış ayracı tek başına bir satırda görünemez. Bazıları, korsenin bir sonraki satırda yaşamasına izin vermek için lexer'in ileriye bakması gerektiğini savundu. Biz anlaşımadık. Go kodunun [gofmt](#) tarafından otomatik olarak biçimlendirilmesi amaçlandığından, bazı stillerin seçilmesi gerekir. Bu stil, C veya Java'da kullandığınızdan farklı olabilir, ancak Go farklı bir dildir ve gofmt'nin stili diğerleri kadar iyidir.

Daha önemli - çok daha da önemlisi - tüm Go programları için tek, programatik olarak zorunlu bir formatın avantajları, belirli bir stilin algılanan dezavantajlarından büyük ölçüde ağır basar. Go'nun stiline, Go'nun etkileşimli bir uygulamasının standart sözdizimini özel kurallar olmaksızın

her seferinde bir satır olarak kullanabileceği anlamına geldiğini unutmayın.

→ Neden çöp toplama yapıyor? Çok masraflı olmayacak mı?

Sistem programlarındaki en büyük muhasebe kaynaklarından biri, tahsis edilen nesnelerin yaşam sürelerini yönetmektir. Manuel olarak yapıldığı C gibi dillerde, önemli miktarda programcı zamanı tüketebilir ve genellikle zararlı hataların sebebidir. Yardımcı olmak için mekanizmalar sağlayan C ++ veya Rust gibi dillerde bile, bu mekanizmalar yazılımın tasarımı üzerinde önemli bir etkiye sahip olabilir ve genellikle kendi başına programlama ek yükü ekleyebilir. Bu tür programcı genel giderlerini ortadan kaldırmanın kritik olduğunu düşündük ve son birkaç yıldaki çöp toplama teknolojisindeki ilerlemeler, ağa bağlı sistemler için uygun bir yaklaşım olabileceğine dair yeterince ucuz ve yeterince düşük gecikme süresiyle uygulanabileceğine dair bize güven verdi.

Eşzamanlı programlamanın zorluğunun çoğunun kökleri nesne yaşam süresi problemine dayanır: nesneler iş parçacıkları arasında geçerken, güvenli bir şekilde serbest kalmalarını garanti etmek zahmetli hale gelir. Otomatik çöp toplama, eşzamanlı kodu çok daha kolay hale getirir yazmak. Elbette, eşzamanlı bir ortamda çöp toplamayı uygulamak başlı başına bir zorluktur, ancak bunu her programda değil bir kez karşılamak herkese yardımcı olur. Son olarak, eşzamanlılık bir yana, çöp toplama arabirimleri daha basit hale getirir çünkü bunlar arasında belleğin nasıl yönetildiğini belirtmelerine gerek yoktur. Bu, Rust gibi dillerde kaynakları yönetme sorununa yeni fikirler getiren son çalışmaların yanlış yönlendirildiği anlamına gelmez; bu çalışmayı teşvik ediyoruz ve nasıl geliştiğini görmekten heyecan duyuyoruz.

Ancak Go, yalnızca çöp toplama ve çöp toplama yoluyla nesne yaşam sürelerini ele alarak daha geleneksel bir yaklaşım benimsiyor. Mevcut uygulama, bir işaret ve süpür toplayıcıdır. Makine çok işlemcili ise, toplayıcı ana programla paralel olarak ayrı bir CPU çekirdeği üzerinde çalışır. Son yıllarda toplayıcı üzerinde yapılan büyük çalışma, büyük yığınlar için bile duraklama sürelerini sık sık milisaniyenin altındaki aralığa indirdi ve bu da çöplere yönelik başlıca itirazlardan birini ortadan kaldırıyor ağa bağlı sunucularda toplama. Algoritmayı iyileştirmeye, ek yükü ve gecikmeyi daha da azaltmaya ve yeni yaklaşımları keşfetmeye yönelik çalışmalar devam ediyor. Go ekibinden Rick Hudson'ın 2018 [ISMM açılış konuşması](#) şimdiye kadarki ilerlemeyi anlatıyor ve bazı gelecekteki yaklaşımlar öneriyor.

Performans konusuna gelince, Go'nun programcıya bellek düzeni ve tahsisi üzerinde, çöp toplama dillerinde tipik olandan çok daha fazla, hatırı sayılır bir kontrol sağladığını unutmayın. Dikkatli bir programcı, dili iyi kullanarak çöp toplama ek yükünü önemli ölçüde azaltabilir; Go'nun profil oluşturma araçlarının bir gösterimi de dahil olmak üzere, çalışılmış bir örnek için [Go programlarının profilini çıkarma](#) hakkındaki makaleye bakın.