Rust Programlama Dili
v1.0

Bu metnin ozgun hali **Rust Toplulugu** uyeleri **Steve Klabnik** ve **Carol Nichols** tarafindan "**The Rust Programming Language**" ismi ile yayimlanmis ve **Ozgur Kara** tarafindan **Turkce** diline yeniden derlenmistir.

Bu metin hazirlanirken **Rust 1.59.0** versiyonu (24 Feb 2022) yayindaydi. Metnin orijinal haline HTML formatinda https://doc.rust-lang.org/stable/book adresinden ya da **Rust** kurulumundan sonra "r**ustup docs-book**" komutunu kosarak erisebilirsiniz.

Bu metnin ebook formatinda bir ornegi ise **No Starch Press** adresinden temin edilebilir.

https://nostarch.com/Rust2018

Cevirmen Notu:

Rust Programlama Dili Turkce cevirisi yapilirken sade bir anlatim dili kullanilmis ve metnin ilerleyen kisimlarinda da goreceginiz uzere Turkce karakterlere yer verilmemis olup bu bilincli bir yaklasimdir. Yine de metinde yer alan bir hata veya ceviri hatasi ile karsilasirsaniz dilerseniz duzelterek kendiniz yeniden yayimlayin ya da cevirmene ileterek guncellenmesini talep edin.

Ceviri hazirlanirken PDF formatinin secilmesi aslinda orijinal dokumanin epub formatinda lisans kosullarina bagli kalmasindan dolayidir. Yine de ileride bir HTML duzenlemesi ya da farkli formatlara gecis icin her sayfanin altina sayfa numarasi ilistirildi.

 Asagida ise bir sonraki versiyonda eklenmesi planlanan ancak hem Turkce karsiliklari hususunda, hem de konunun sade anlatimi adina zaman gerektiren kisimlar yer almaktadir. Bir cesit metin surum yonetimi olarak dusunebilirsiniz.

Ozgur Kara
Dublin | 2022

Metin Surum Yonetimi

Guncel surum: v1.0
Sonraki surum: v1.1 (ilk rakam major degisimi ikinci rakam minor degisimi ifade eder)

Eklenecek Konular:

1. Enums
2. Pattern Matching
3. Packages and Crates
4. Writing Automated Tests
5. Smart Pointers

Onsoz:

Rust programlama dili oncesinde paradigmalar her zaman bu kadar net degildi, Rust Programlama dili temelde yetkilendirme ile ilgilidir ve su anda ne tur bir kod yaziyor olursaniz oolun, Rust daha uzaga ulasmaniza, daha once yaptiginizdan daha genis bir alana yonelmenize kisaca limitleri asmaniza ve limitleri asarken guvenli bir programlama yapabilmenize olanak saglar.

Ornek olarak, bellek yonetimi, verinin ciktisi ve ayni zamanda eszamanlilik konusunu dusuk duzeyde ve ayrintilariyla ilgilenerek yani kisaca “sistem seviyesinde” bir goz atacak olursak geleneksel olarak programlamanin ve programlarin calismasinin bu goruntusu gozunuze gizemliymiscesine gorunur. Ancak yillarini kotu giden bir program akisinin arkaplanina bakmaya adayan ve tuzaklardan kacinan seckin birkac insan tarafindan bu gizemlilik ortadan kaldirilir. Bunu yapabilen programcilar dahi hala gelistirdigi programlarin istismarlara ve cokmelere karsi direnc gosterebilmesi icin dikkatli bir sekilde programlarini gelistirmeye calisirlar.

Rust Programlama Dili bu geleneksel ve eski tuzaklari ortadan kaldirarak programlama yolculugunda size tamamen guvenli bir sekilde yardimci olacak pek cok suslenmis araclar kullanarak karsilasilmasi beklenen engelleri ortadan kaldirmayi hedefler. Basta da bahsetmis oldugumuz dusuk seviye yani sistem seviyesinde karsilasilacak engelleri normalde asabilmek icin efor sarfetmesi gereken programcilar, Rust Programlama Dili kullanarak bu geleneksel risk ve engellerden ornegin cokmelerden ve bunun beraberinde gelen guvenlik aciklari ve risklerden kacinabilir ve bunu yaparken herhangi bir ince ayrintiya takilmadan yola devam edebilirler.

Halihazirda dusuk seviyeli programlar dedigimiz programlama dilleri ile gelistirme yapan programcilar, bu hedeflerine ulasabilmek icin Rust Programlama Dili’ni kullanabilir. Ornegin. Rust Programlama Dili’nde paralellik ya da paralel programlama metodojilerini uzerinde calistiginiz projeye eklemeniz dusuk risk tasiyan bir islem olsa da Derleyici sizin icin klasik hatalari yine de yakalayacaktir. Beklenmeyen kesmeler ve cokmelerle beraber guvenlik aciklari olmayacagina guvenerek kodunuzdaki daha agresif optimizasyonlarin ustesinden gelebilirsiniz.

Fakat Rust Programlama Dili, bu metodojiler kullanilirken dusuk seviyeli sistem programlama ile sizi sinirli birakmayacaktir. CLI (komut satiri) uygulamalari ya da web sunuculari gibi diger bircok kod turunu yazmayi oldukca keyifli kilacak kadar etkileyici ve ergonomik imkanlar sunacaktir.

Bu metin, Rust Programlama Dili’ni kullanan programcilarin daha guclu olan potansiyel yonlerini tamamen benimseyerek bunu yazdiklari koda aktarabilmesine imkan saglar. Bu ayni zamanda sadece Rust dilini ogrenmenize degil genel olarak yazilim gelistirme konseptinde programci olarak karsilasacaginiz erisim ve guvenlik handikaplarini asmanizda yardimci olmayi amaclayan samimi bir dokumandir.

Hazirsaniz korkmayin ve derinliklere dalmaya hazir olun. **Rust Toplulugu**’na hos geldiniz!

-- **Nicholas Matsakis** ve **Aaron Turon**

Cevirmenin Notu:
Bu metnin bundan sonraki geri kalaninda Rust Programa Dili kisaca Rust olarak anilacaktir.

Giris:

Rust hakkinda bir giris kitabi olan Rust Programlama Dili'ne hos geldiniz. Rust Programlama Dili, daha hizli ve daha guvenilir programlar ve yazilimlar gelistirmenize yardimci olacak sekilde tasarlanmistir. Yuksek seviyeli ergonomi ve dusuk seviyeli kontroller bir programlama dili tasariminda genellikle celiskiler icerir;

Rust ise bu celiskilerin catismasina meydan okumaktadir. Guclu teknik kapasiteyi ve harika bir gelistirici deneyimini dengeleyen Rust, size dusuk seviyeli sistem ayrintilarini (ornegin bellek kullanimi) bu tur kontrollerle geleneksel olarak iliskilendirilen tum guclukleri ortadan kaldirarak kontrol etme secenegi sunar.

Rust kimler icindir?

Rust, tasidigi pek cok karakteristik ve yapisal icerigi nedeni ile aslinda bircok gelistirici ve programci icin ideal olmasina ragmen bu icerikleri referans olarak en onemli bazi gruplardan birkacina goz atalim.

Ekip halinde calisan gelistiriciler:

Rust, farkli duzeylerde sistem programlama tecrubesine sahip gelistiricileren olusan ekipler arasinda isbirligi yapmak icin uretken bir arac oldugunu kanitliyor. Dusuk seviyeli kodlar, diger bircok dilde yalnizca deneyimli gelistiriciler tarafindan kapsamli testler ve dikkatli kod incelemesi yoluyla yakalanabilen cesitli ince hatalara egilimlidir. Rust'a derleyici, eszamanlilik hatalari da dahil olmak uzere bu anlasilmasi zor hatalarla kod derlemeyi reddederek bir kapi bekcisi rolu eklenmistir. Derleyici ile birlikte calisarak ekip, zamanlarini hatalarin pesine dusmek yerine programin mantigina odaklanarak gecirebilir ve zaman kaybini tamamen onlemis olur.

Rust bunlardan baska olarak cagdas bir yazilim gelistirme anlayisina uygun sekilde bazi araclari programlama dunyasina kazandirir ve programlama paradigmalarinin yonunu degistirir:

Cargo: Bagimlilik yoneticisi ve kod olusturma araci. Rust ile gelistirme yaparken Cargo aracini kullanarak bagimliliklarinizi ve bunlarin derlemesini ve yonetilmesini sorunsuz ve tutarli bir sekilde gerceklestirebilirsiniz.

Rustfmt: Ekip halinde calisan gelistiriciler arasinda tutarli bir kodlama stili benimsenmesini saglar. Rust Dil Sunucusu (Rust Language Server) kod tamamlama ve satir ici hata mesajlari icin Yerlesik Gelistirme Ortami (Integrated Development Environment) entegrasyonunu da destekler.

Ekip halinde calisan gelistiriciler Rust ekosistemindeki bu ve benzer diger araclari kullanarak, sistem duzeyinde kod yazarken daha uretken olabilirler.

Ogrenciler:

Rust, sistem programlama kavramlarini ve yazilim paradigmalarini ogrenmek isteyen ogrenciler icin de uygundur. Ornegin Rust'i kullanarak bircok kisi isletim sistemi gelistirme gibi konulari da ogrenmis oldu. Rust Toplulugu sicakkanli ve ogrencilerin her zaman sorularini yanitlamaktan mutluluk duyuyor. Rust ekipleri, bu metin ve kitap gibi cabalarla kavramlari ve gelistirme asamalarini derinlemesine ele alarak yeni baslayanlar icin daha erisilebilir bilgiler sunmayi her zaman hedeflemektedir.

Sirketler:
Buyuk ve kucuk olcekte yuzlerce sirket, cesitli gorevler icin uretimde Rust kullanmaktadir. Bu gorevler arasinda komut satiri araclari, web hizmetleri, DevOps araclari, gomulu (embedded) cihazlar, ses ve video analizi ve kod donusturme, kripto para birimleri, biyoinformatik, arama motorlari, nesnelerin interneti (IoT), makine ogrenimi (machine learning) ve hatta Firefox web tarayicisinin buyuk bolumleri de dahil pek cok farkli gelistirme surecleri yer almaktadir.

Acik kaynak gelistiricileri:
Rust, Rust toplulugu, gelistirici araclari ve kitapliklar olusturmak isteyenler icin de idealdir. Rust Programlama Dili'ne katkida bulunmanizi cok isteriz.

Hiz ve Kararlilik Tereddutunde olan Insanlar:
Rust, bir dilde hiz ve istikrar isteyen insanlar icin oldukca uygundur. Hiz derken, Rust ile olusturabileceginiz programlarin hizini ve Rust'in bunlari yazmaniza izin verdigi hizi kastediyoruz. Rust derleyicisinin denetimleri, ozellik eklemeleri ve yeniden duzenleme yoluyla kararlilik saglar. Bu, gelistiricilerin genellikle degistirmekten korktuklari, bu kontrollerin olmadigi dillerdeki kirilgan eski kodlama stilinin aksine sifir maliyet iceren soyutlamalar ve manuel olarak yazilan kod parcalari kadar hizli bir sekilde daha dusuk seviyeli koda derlenen daha yuksek seviyeli ozellikler icin cabalayarak guvenli kodun da hizli bir kod parcasi olmasina olanak saglar.

Rust, diger bircok kullanici da desteklemeyti umuyor; burada bahsedilenler sadece en buyuk ortak paydaslardan bazilaridir. Genel olarak, Rust'in en buyuk amaci, guvenlik ve uretkenlik, hiz ve ergonomi saglayarak programcilarin on yillardir kabul ettigi odunleri ortadan kaldirmaktir.

Rust'i deneyin ve sundugu seceneklerin sizin icin ise yarayip yaramadigini gorun.

Bu kitap kimler icindir?
Bu kitap baska programlama dilinde kod yazdiginizi varsayiyor olsa da hangi dili kullandiginizla ilgili bir varsayima sahip degildir. Bu kitaptaki metinleri cok cesitli programlama gecmisine sahip gelistiricilerin genis capta yararlanabilir halde olmasi icin cabaladik. Programlamanin ne oldugu veya onun hakkinda nasil dusunulecegi hakkinda konusmak icin cok zaman harcamadik. Bu nedenle programlama konusunda tamamen yeniyseniz, bunun oncesinde bir **"Programlamaya Giris"** gibi bir kitabi okuyarak baslangic yapabilirsiniz.

Bu kitap nasil kullanilir?
Bu kitabin hazirlanis sekli kitabi ilk bolumlerinden son bolumlerine dogru okudugunuzu varsayacak sekilde tasarlanmistir. Bu nedenle sonraki bolumler onceki bolumlerde yer alan kavramlar uzerine insa edilmistir ve onceki bolumlerde ayrintisi verilmeyen bir konu daha sonraki uygun bolumde tekrar ele alinmis olabilir.

Bu kitapta iki tur bolum bulacaksiniz: Kavramsal Bolumler ve Proje Bolumleri. Konsept Bolumunde ise Rust'in yaklasimsal yonu ele alinir. Proje bolumlerinde ilgili bolumleri ve ogrendiklerinizi uygulamaya ve kucuk program ornekleri ile yetkinliginizi kanitlamaya baslayacaksiniz. Bolum 2, 12 ve 20 bolumleri Proje bolumleridir, geri kalani ise konsept bolumleridir.

Baslarken:

Rust yolculuguna bir yerden baslamaliyiz. Her yolculuk bir noktadan baslar. Rust icinse kurulum ve geleneksel bir **“Merhaba Dunya!”** uygulamasi gelistirerek ve Cargo uygulamasini kullanarak paket yoneticisine bir bakis atip iyi bir baslangic yapabiliriz.

Kurulum:

Rust ogrenmek ve kullanmak icin ilk adim Rust kurulumunu gerceklestirmektir. Rust’i Rust surumlerini ve ilgili araclari yonetmek icin bir komut satiri araci olan “**rustup”** kullanarak gerceklestirecegiz. Ancak bunun icin bir internet baglantisina ihtiyaciniz oldugunu unutmayin.

Not: Herhangi bir sebeple rustup kullanmamayi tercih ederseniz Rust’in rustup kullanmadan kurulum yonergelerini iceren web sayfasina goz atabilirsiniz.

https://forge.rust-lang.org/infra/other-installation-methods.html

Asagidaki adimlar, Rust derleyicisinin en son kararli surumunu yukler. Rust’in kararlilik garantisi, bu kitaptaki tum orneklerin daha yeni Rust surumleriyle de uyumlu derlenmesine olanak saglar. Cikti surumler arasinda bazi farkliliklar gosterse de Rust genellikle hata mesajlarini ve uyarilari iyilestirecektir. Baska bir deyisle, bu adimlari kullanarak kurdugunuz her yeni ve kararli Rust surumu bu kitabin icerigi ile beklendigi sekilde calismalidir.

Komut Satiri Uyarisi:

Bu ve gelecek bolumlerde terminalde kullanilan bazi komutlar goreceksiniz. Bir terminalde girmeniz gereken komutlarin tamami **$** isareti ile baslar ancak **$** karakterini yazmaniz beklenmez, Bu **Windows PowerShell** icin **$** yerine > isareti ile isaretlenmistir.

Linux ve macOS icin yukleme:

Linux ve macOS kullaniyorsaniz, bir terminal (komut satiri uygulamasi) acin ve asagidaki komutu girin:

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Bu komut Rust kurulum dosyasini indirir ve Rust’in en son kararli surumunu yukleyen rustup aracinin da kurulmasini saglar. Kurulum basarili oldugunda asagidaki satiri goreceksiniz:

```
Rust is installed now. Great!
```

Ayrica kurulumu tamamlamak icin linker adi verilen baglayici programina da ihtiyaciniz olacak. Eger buna benzer bir hata alirsaniz C derleyicisi yuklemelisiniz. Ornegin macOS kullaniyorsaniz asagidaki komutla bir C derleyicisi yukleyebilirsiniz:

```
$ xcode-select –install
```

Linux kullaniyor iseniz GCC veya Clang kurabilirsiniz. Ornegin Debian veya Ubuntu kullanicilari build-essential paketini kurarak bunlara sahip olabilir.

Guncelleme ya da Kaldirma:

Rust'i rustup programini kullanarak yukledikten sonra en son surume guncellemek kolaydir. Komut satirindan asagidaki komutu cagirin:

$ rustup update

Eger Rust kurulumunu kaldirmak istiyorsaniz asagidaki komutu kullanabilirsiniz:

$ rustup self uninstall

Sorun giderme:

Rust kurulumunun dogru olup olmadigini kontrol etmek icin asagidaki komutu calistirin:

$ rustc –version

Tum adimlari dogru yaptiniz ancak Rust calismiyorsa yardim almak icin Rust Discord sunucusundaki **#beginners** kanalina gidebilirsiniz. Orada, size yardimci olacak diger Rustacean'larla tanisabilirsiniz.

Rustacean: Rust toplulugu uyelerinin kendisine taktigi takma ad. Tebrikler artik siz de bir **Rustacean**'siniz.

Yerel Belgeler:

Rust kurulumu icerisinde ayrica yerel olarak (internet baglantisina gerek kalmadan) faydalanabileceginiz bir cevrimdisi dokumantasyona sahip olacaksiniz. Tarayicinizda bu yerel belgelere ulasmak icin **"rustup doc"** komutunu calistirabilirsiniz. Ayni zamanda standart kutuphane (standard library) tarafindan saglanan bir tur veya islevle ilgili ne ise yaradigindan emin olmadiginiz bir cikti ile karsilasirsaniz bunu arastirmak ve ogrenmek icin uygulama programlama arabirimi (API) belgelerine goz atabilirsiniz.

Merhaba, Dunya!

Artik Rust'i yuklediginize gore ilk Rust programinizi yazabilirsiniz. Yeni bir dil ogrenirken **Merhaba, Dunya!** metnini ekrana yazdirmak geleneksel bir yaklasimdir. Rust diline ilk giriste de bunu yapacagiz!

Bir Proje Dizini Olusturma:

Rust ile yazdiginiz kodlari saklamak icin bir dizin olusturmak iyi bir baslangic yontemidir. Bununla beraber Rust kodunuzun nerede oldugu pek onemli degildir ancak yine de bu dokumandaki tum ornek calismalar icin bir proje dizini olusturmanizi ve tum projeleri orada tutmanizi tavsiye ederiz.

Yeni bir klasor olusturmak icin asagidaki komutu kosun ve Merhaba, Dunya! uygulamaniz icin bir proje klasoru yaratin:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir merhaba_dunya
$ cd merhaba_dunya
```

Not: Yukarida da bahsetmis oldugumuz gibi bu bir ornektir ve bir gereklilik degildir. Projenizi ve kodlarinizi kendinize ozgu sekilde de saklayabilir ve farkli isimler de verebilirsiniz.

Simdi yeni bir kaynak dosyasi (source file) olusturun ve **main.rs** olarak adlandirin. Rust programlama kaynak kodlarini iceren dosyalar her zaman **".rs"** uzantisi ile biter. Dosya adinizda birden fazla kelime varsa bunlari ayirt etmek icin alt cizgi (_) kullanabilirsiniz.

Simdi bu kaynak dosyanizin icerisine asagidaki kod parcaciklarini yazin ve sonra kaydederek dosyadan cikis yapin:

```
fn main() {
    println!("Merhaba, Dunya!");
}
```

Simdi ise bu kaynak kodu dosyanizi Rust derleyicisini kullanarak derlemek (compile) icin asagidaki komutu kullanin:

$ rustc main.rs

Artik derlenmis ve calistirilabilir ilk Rust programiniza sahipsiniz. Programinizi calistirmak icin asagidaki komutu kullanin:

$ ./main

Merhaba, Dunya!

Tebrikler! Hersey yolunda gitti ise programinizi calistirdiktan sonra terminalde **“Merhaba, Dunya!”** mesajini gormus olmalisiniz. Eger bu mesaji gormuyorsaniz **“Sorun Giderme”** adimlarina bakmalisiniz.

Bu artik sizi bir Rust Programcisi (Rustacean) yapar. Aramiza hos geldiniz!

Bir Rust Programinin Anatomisi:

“Merhaba, Dunya!” programinizda neler oldugunu ayrintili olarak gozden gecirelim. Iste yapbozun ilk parcasi:

```
fn main () {

}
```

Bu satirlar Rust’ta bir fonksiyon islevini tanimlayan bloklardir ve ana fonksiyon yani **“main ()”** ozel bir tanima sahiptir. Her zaman bir Rust programinin yurutulmesinde calisan ilk fonksiyondur. Bu fonksiyon parametresi olmayan ve hicbir sey dondurmeyen bir fonksiyondur. Eger parametreler olsaydi parantezler **“()”** icinde yer alirlardi.

Ayrica bu main fonksiyonun govdesinin kume parantezlerine **“{}”** (brackets) sarildigina dikkat etmelisiniz. Rust tum fonksiyonlarda bu tur fonksiyon govdesi yapisina ihtiyac duyar. Aralarina bosluk birakarak fonksiyon bildirimini ayni satira yerlestirmek iyi bir kodlama stili olarak bilinir.

Eger Rust ile gelistirdiginiz projelerde standart bir stile bagli kalmak istiyorsaniz kodunuzu belirli bir stille bicimlendirmek icin **“rustfmt”** aracini kullanabilirsiniz. Rustfmt rustc derleyicisi gibi standart Rust kurulumuna dahil edilmistir.

Bu ana (main) fonksiyonun icerisinde println satirinin yer aldigini gordunuz. Bu satir, bu kucuk Merhaba, Dunya! Programinizdaki tum ana isi yurutur: verdiginiz metni ekrana yazdirir. Burada bazi dikkat edilmesi gereken detaylar yer alir:

Oncelikle, Rust stili bir sekme ile degil dort boslukla girinti yapmaktadir.

Ikincisi, **println!** Bir Rust makrosu cagirir. Bunun yerine eger bir baska fonksiyon cagirsaydi sadece **println** (! unlem isareti olmadan) cagrilirdi. Rust makrolarini ileriki bolumlerde ayrintili olarak tartisiyor olacagiz. Simdilik normal bir **println** cagrisi yerine **println!** gibi makro cagirdiginizi ve makrolarin her zaman fonksiyonlarla ayni kurallara uymadigini goz onunde bulundurmalisiniz.

Son olarak **“Merhaba, Dunya!”** bir karakter katari (string) ve bu karakter katari yiginini **println!** makrosuna arguman olarak iletiyoruz ve ana (main) fonksiyonumuz calistiginda println! cagriliyor ve sonucunda ekranda **“Merhaba, Dunya”** mesajini goruyorsunuz.

Ayrica bir satirin sonunda noktali virgul (;) kullanildigini unutmayin. Noktali virgul, ifadenin bittigini ve bir sonrakinin eger varsa baslamaya hazir oldugunu derleyiciye iletir. Kisaca Rust kodunun cogu satiri noktali virgul ile bitecektir.

Derleme ve Calistirma:

Yukarida da gordugunuz ve test ettiginiz uzere derleme ve calistirma adimlari ayri adimlardir. Bu yuzden once programinizi rustc kullanarak derlediniz ve ardindan yurutulebilir dosyayi (executable file) calistirdiniz. Simdi bu surecteki adimlari inceleyelim.

Bir Rust programini calistirmadan once, Rust derleyicisini kullanarak yani rustc komutunu cagirarak ve kaynak kodu dosyanizin adini asagidaki sekilde ona ileterek derleme islemini yapmalisiniz:

$ rustc main.rs

Daha onceden C veya C++ gibi diller kullandi iseniz bunun GCC veya Clang derleme adimlarina benzedigini farketmissinizdir. Basarili bir derleme isleminden sonra Rust derleyicisi yurutulebilir bir dosya olusturur. Bu dosya ayni zamanda ikili (binary) yurutulebilir bir dosyadir.

Oncesinde Ruby, Python ya da JavaScript gibi diller kullandi iseniz bir programin ayri adimlar olarak derlenmesine aliskin olmayabilirsiniz. Bu diller yorumlayici (interpreter) dillerdir ve satir satir yorumlanarak calistirilir. Rust ise onceden derlenmis bir dildir, yani bir programi derler ve yurutulebilir dosyayi baska birine verirseniz onlar Rust kurulu olmadan dahi programi calistirabilirler.

Ornegin bir arkadasiniza .rb ya da .py uzantili kaynak kodu dosyasi verirseniz o arkadasiniz, bu kaynak kodlarini calistirmak icin Ruby ya da Python uygulamalarinin da kurulu olmasini gerektirecektir. Ancak Rust dilinde programinizi derlemek ve calistirmak icin yalnizca bir komuta ihtiyaciniz vardir. Dil Tasarimi denilen bu yaklasimda her sey bir degis tokustur.

Basit programlar icin sadece rustc ile derlemek iyidir, ancak projeniz buyudukce tum secenekleri yonetmek ve kodunuzu paylasmayi kolaylastirmak isteyeceksiniz. Ardindan, sizi gercek dunyadaki Rust programlari yazmaniza yardimci olacak **“Cargo”** ile tanistiracagiz.

Merhaba, Cargo!

Cargo Rust programlama dilinin yapi araci (build tool) ve paket yoneticisidir (package manager). Cogu Rustacean, Rust projelerini yonetmek icin bu araci kullanirlar cunku Cargo, kodunuzu olusturmak, kodunuzun bagli oldugu kitapliklari (library) indirmek ve bu kitapliklari olusturmak gibi bircok gorevi sizin yerinize gerceklestirir (Eger bagimliliga ihtiyac duyan bir program yazdi iseniz).

Simdiye kadar yazdigimiz en basit Rust programlarinin herhangi bir bagimliligi yoktu. Yani “Merhaba, Dunya!” Cargo ile projelendirdiginizde, yalnizca Cargo’nun kodunuzu olusturmayi yoneten bolumunu kullanir. Daha karmasik Rust programlari yazdikca, bagimliliklar ekleyeceksiniz ve Cargo kullanarak bir projeye baslar iseniz bagimliliklari ekleme ve cikarma konusunda daha kolay bir yonteme sahip olursunuz.

Rust projelerinin buyuk cogunlugu Cargo kullandigindan, bu dokumanin geri kalaninda sizin de Cargo kullandigini varsayiyor olacagiz. “Kurulum” bolumunde belirtilen resmi yukleyiciyi kullanarak Rust kurdu iseniz Cargo Rust kurulumu ile beraber yuklenmistir. Eger Rust’i baska yontemlerle kurdu iseniz asagidaki komutu kullanarak Cargo kurulumunun yapilip yapilmadigini kontrol edebilirsiniz:

$ cargo –version

Cargo Aracini Kullanarak Rust Projesi Olusturma:

Simdi ise Cargo aracini kullanarak yeni bir proje olusturalim ve “Merhaba, Dunya!” projemizi bu projeye entegre edelim:

$ cargo new merhaba_dunya_cargo

$ cd merhaba_dunya_cargo

Ilk komutunuz merhaba_dunya_cargo adinda yeni bir klasor olusturdu. Projenize de merhaba_dunya_cargo adini vermis oldunuz ve Cargo araci bu yeni klasoru proje adinizla ayni isimde olusturdu.

Sonrasinda merhaba_dunya_cargo dizinine gittiginizde doyalari listeleyin. Cargo’nun sizin icin iki dosya ve bir dizin olusturdugunu goreceksiniz. Bunlar sunlardir:

Cargo.toml (dosya)

src (dizin)

src/main.rs (dizin icerisinde main.rs ana program dosyasi)

Ayni zamanda bir .gitignore dosyasi da olusturarak bir GIT (versiyon kontrol yazilimi) deposu da olusturmus oldu. Mevcut bir GIT deposunda yeniden cargo aracini cagirirsaniz tekrar GIT dosyalari olusturulmaz; bunu engellemek isterseniz asagidaki komutu kullanin:

$ cargo new –vcs=git

Herhangi bir metin editoru ile Cargo.toml dosyasini acin. Muhtemelen asagidaki sekilde goruntulenmelidir.

```
[package]
name = "merhaba_dunya_cargo"
version = "0.1.0"
edition = "2022"
[dependencies]
```

Bu dosya Cargo'nun konfigurasyonlarini iceren bir formata sahip sekilde TOML (Tom's Obvious Minimal Language) formatindadir.

Ilk satir [package] icerir ve bu konfigurasyon dosyasinin bir paketi yapilandirmak icin kullanildigini bildirir. Sonraki satirlar, Cargo'nun programinizi derlemesi icin ihtiyac duydugu yapilandirma bilgilerini saklar. Bu bilgiler, programin adi (name), versiyonu (version) ve baski (edition) bilgileridir.

Son satirda ise [dependencies] satirinin yer aldigini gormussunuzdur. Projenizin bagimliliklarindan birini eklemeniz icin baslangici isaret eder. Rust programlama dilinde kod paketlerine sandik **"crate"** denir. Bu proje icin simdilik baska sandiklara ihtiyac duymayacaksiniz ancak bir sonraki projede ihtiyaciniz olacak ve bu yuzden bagimliliklar konusunu gelecek bolumde irdeleyecegiz.

Eger src/main.rs dosyasina goz atarsaniz Cargo aracinin sizin icin bir "Merhaba, Dunya!" (Hello, World!) kod blogunu da ornek olarak ekledigini gorursunuz. Tipki sizin daha once yazmis oldugunuz program gibi! Sizin yazdiginiz bir onceki programla Cargo tarafindan olusturulan proje arasindaki fark Cargo'nun kaynak kod dosyasini src (source) dizini altinda yerlestirmesi ve ust dizinde Cargo.toml adinda bir konfigurasyon ve yapilandirma dosyasini da olusturmus olmasidir.

Gorundugu uzere Cargo kaynak kod dosyalarinizin src dizini icinde yasamasini bekler. En ust duzey proje dizini ise belki BeniOku ya da Lisans bilgilerini iceren kodunuzun calismasina etki etmeyen diger tum her sey icin kullanilabilir. Cargo'yu kullanmak projelerinizi hiyerarsik bir sekilde duzenlemenize de yardimci oolur. Her seyin bir yeri var ve her sey yerli yerinde!

Bu arada daha once yaptigimiz gibi bir "Merhaba, Dunya" programi yazdi izeniz ve Cargo aracini kullanmadi iseniz daha sonrasinda Cargo paket yapisini kullanan bir sekle donusturmek kolaydir. Proje kaynak kodlarinizi src/ dizinine tasiyin ve uygun bir Cargo.toml dosyasi olusturun.

Cargo Projesi Olusturma ve Yurutme:

Simdi ise Cargo ile beraber yeni "Merhaba, Dunya!" kaynak kodunu yapilandirip (build) calistirarak nelerin farkli olduguna goz atalim. Cargo aracini kullanarak proje dizininizde iken su komutu kosun:

$ cargo build

Bu komut proje dizininiz altinda target/ adinda bir klasor yaratarak bunun icerisinde merhaba_dunya_cargo adinda yurutulebilir bir ikili (binary) dosya olusturur ve programinizi su sekilde calistirabilirsiniz:

$ ./target/debug/merhaba_dunya_cargo

Hello, World!

Her sey yolunda giderse Merhaba, Dunya! yani Cargo ile gelen Hello, World! Cagirilarak ekrana yazdirilacaktir. Ayrica Cargo araci ile insa etmek (build) proje dizininin en ustunde Cargo.lock adindan bir dosyanin da olusturulmasini saglar. Bu dosya, projenizdeki bagimliliklarin tam surumlerini takip eder. Ornek uygulamanin bagimliliklari olmadigi icin su anda daha az detay iceren bir dosya icerigi goreceksiniz.

Ayni zamanda Cargo aracini kullanarak programinizi yurutebilirsiniz. Bunun icin asagidaki komutu kullanin ancak uyari mesajlarinda derleme yapmadigina dikkat edin cunku Cargo daha once insa edilmis bir yurutulebilir dosya oldugunu gorecek ve onu kopyalayacaktir. Ancak kaynak kodunuzda degisiklik yaparsaniz bu defa once derleyecek aslindan bu derleyici tarafindan derlenmis yurutulebilir dosyayi ana dizininizde hazirlayacaktir.

$ cargo run

Finihed 0.03 s

Running target/debug/merhaba_dunya_cargo

Hello, World!

Cargo araci ayni zamanda bir kontrol parametresine de sahiptir (check). Bu parametre ile Cargo cagirirsaniz kaynak kodu dosyaniz derlenir ancak yurutulebilir bir dosya uretilmeden kodunuz hizlica kontrol edilir.

$ cargo check

Goruldugu uzere cogu zaman Cargo ile kontrol (**cargo check**) bir yurutulebilir dosya uretme adimini atladigi icin **cargo build** komutundan cok daha hizli olacaktir. Kod yazarken surekli programinizi kontrol etmeniz gerekiyorsa **cargo check** bu surece yardimci olacaktir.

Cargo hakkinda simdiye kadar ogrendiklerimizin bir ozeti sunlardir:

cargo build kullanarak bir projeyi insa edebiliriz.

cargo run komutunu cagirarak tek seferde programimizi calistirabiliriz.

cargo check komutunu kullanarak yurutulebilir bir dosya olusturmadan hatalari kontrol edebiliriz.

Cargo derleme sonrasinda kodumuzla ayni dizinde olmak yerine target/ hedef dizini icinde yurutulebilir dosyalarimizi saklar.

Cargo kullanmanin bir diger avantaji hangi isletim sistemini kullaniyor olursaniz olun komutlarin her zaman ayni olmasidir. Bu nedenle Linux, macOS, SunOS ve Windows icin ozel talimatlara ihtiyac yoktur.

Yayinlamak icin Yeni bir Surum Olusturmak:

Projeniz nihayet yayinlanmaya hazir oldugunda onu optimizasyonlarla derlemek icin asagidaki komutu kullanabilirsiniz:

$ cargo build –release

Bu komut hata ayiklama (debugging) dizini (target/debug) yerine surum dizininde programinizin yurutulebilir dosya ornegini olusturacaktir. Optimizasyonlar, Rust kodunuzun daha hizli calismasini saglar, ancak bunlari acmak programinizin derlenmesi icin gereken sureyi de uzatir.

Bu nedenle iki farkli profil vardir. Birisi hizli surum olusturma, digeri ise tekrar tekrar yeniden olusturma yerine olabildigince hizli ve siklikla olusturma ve son kullaniciya (end users) teslim edeceginiz surumu olusturmadir. Kodunuzun calisma surelerini kiyaslamak icin (benchmarking) target/release altindaki yurutulebilir dosyayi kullandiginizdan emin olmalisiniz.

Cargo Yapilandirma (Build and Convention) Sozlesmesi:

Basit projelerle Cargo araci sadece rustc derleyici kullanmaktan cok fazla bir deger saglamamis gibi gorunebilir ancak programlariniz daha karmasik hale geldikce Cargo Sozlesmesi degerini kanitlar. Birden fazla sandiktan olusan daha karmasik projelerde Cargo’nun tum kod yapinizi koordine etmesine izin vermeniz oldukca kolay olacak.

Merhaba_Dunya_Cargo projesi basit olsa da, Rust kariyerinizin geri kalaninda kullanacaginiz gercek araclarin cogunu barindirir. Bunlardan birisi de GIT versiyon kontrol aracidir.

GIT Kullanimi:

Rust projelerinizde GIT versiyon ve surum kontrol aracini da kullanabilirsiniz. Bunun icin asagidaki komutla ornek bir projeyi deneyin:

$ git clone ornek.com/merhabadunyaprojesi

$ cd merhabadunyaprojesi

$ cargo build

Ozet:

Rustecean olma yolculugunda simdiden harika bir baslangic yaptin! Bu bolumu ozetleyecek olursak:

Rust'in en son kararli surumunu rustup kullanarak yukleyebilirsiniz.

Yerel olarak calisan belgeleri rustup doc komutu ile acabilirsiniz.

Cargo kullanarak yeni proje olusturabilir ve projenizi bagimliliklari ile surume donusturebilirsiniz.

Bunlar harika!

Simdi bir sonraki bolume gecis yapiyoruz. Bu bolumde programlama paradigmalarinin Rust nezdinde nasil calistigini ogrenecegiz.

- BOLUM 2 -

Ortak Programlama Kavramlari:

Bu bolumde, hemen hemen her programlama konseptinde gormus oldugunuz kavramlari ve bunlarin Rust'ta nasil calistigini goreceksiniz. Diger bircok programlama dilinin ozunde pek cok ortak nokta vardir. Bu nedenle bu bolumde sunulan kavramlarin hicbiri Rust'a ozgu degildir, ancak bunlari Rust baglaminda ele alacagiz ve bu kavramlarin kullanimina iliskin kurallara goz atacagiz.

Ozellikle degiskenler, temel veri turleri, islevler, yorum satirlari ve kontrol akislari hakkinda bilgi ediniyor olacaksiniz. Bu temeller her Rust programinda yer alacak olup bunlari erkenden ogrenmek size baslangic icin guclu bir temel saglayacaktir.

Anahtar Kelimeler: Rust programlama dilie diger dillerde oldugu gibi yalnizca dil tarafindan kullanilmak uzere ayrilmis bir dizi anahtar kelimeyi yapisinda barindirir. Bu kelimelkeri degisken veya fonksyion olarak kullanamazsiniz. Anahtar kelimelerin cogunun ozel anlamlari oldugu gibi Rust bu anahtar kelimeleri cesitli gorevleri yapmak icin kullanir; bu anahtar kelimelere goz atmak icin "EK A" belgesine goz atin.

Degiskenler ve Degismezlik (Immutability):

Degiskenler ve degiskenlerle beraber degerleri saklama bolumunde de belirtildigi uzere varsayilan olarak degiskenler degismezdir (immutability). Bu, Rust'in size sundugu guvenlik ve kolay eszamanliliktan yararlanarak kodunuzu yazmaniz icin size verdigi bircok durtuden biridir. Ancak yine de degiskenlerinizi degistirebilir yapma seceneginiz oldugunu aklinizin bir kosesinde tutmakta yarar vardir.

Rust'in sizi degismezligi tercih etmeye nasil ve neden tesvik ettigini ve bazen bundan neden vazgecmeniz gerekecegini beraber kesfedelim.

Bir degisken degismez oldugunda, bir deger bir ada baglandiktan sonra bu degeri artik degistiremezsiniz. Bunu orneklemek icin Cargo aracini kullanarak yeni bir proje dizini olusturalim. Ardindan bu proje dizininde main.rs kaynak kodu dosyanizi acin ve icerigini asagidaki kodla degistirin.

Degistirdiginiz ve kaydedip ciktiginiz bu kod henuz derlenmeyecek, once degismezlik hatasini ele alacagiz.

Simdi yukarida da bahsettigimiz gibi bir proje olusturalim ve src/main.rs kaynak kodu dosyamizi degiselim:

$ cargo new mutability

$ cd mutability

$ cat src/main.rs

```
fn main() {
    let x = 5;
    println!("x degiskeni icin atanan deger: {}", x);
    x = 6;
    println!("x degiskeninin aldigi yeni deger: {}", x);
}
```

$ cargo run

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("x degiskeni icin atanan deger: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `mutability` due to previous error
```

Bu ornekte, derleyicinizin programlarinizdaki hatalari bulmaniza nasil yardimci oldugunu gormektesiniz. Derleyici hatalari can sikici olabilir, ancak gercekte bunlar yalnizca programinizin henuz yapmak istediginiz seyi guvenli bir sekilde yapmadigi anlamina gelir, bu iyi bir programci olmadiginiz anlamina gelmez. Tecrubeli Rustacean'lar hala derleyici hatalari almaya devam ediyor.

Hata mesaji, hatanin nedeninin degismez (immutability) olarak tanimlanmis bir x degiskenine ikinci bir deger atanmasina calisildigini ancak degismez (immutability) x degiskenine iki kez deger atayamaz oldugunuzu gosterir.

Degismez olarak belirlenmis bir degeri degistirmeye calistigimizda derleme zamaninda hatalar almamiz onemlidir cunku bu durum program gercek ortamda calistirildiginda kritik hatalara yol acabilir. Kodumuzun bir kismi, bir degerin asla degismeyecegi varsayimi ile calisiyorsa ve kodumuzun baska bir kismi da bu degeri degistirmek istiyorsa kodun ilk kisminin tasarlandigi seyi yapmamasi gayet normaldir. Bu tur bir hatanin nedenini, ozellikle ikinci kod parcasi degeri yalnizca bazen degistirdiginde kisaca olaydan (event) sonra bulmak zor olabilir. Rust derleyicisi, bir degerin degismeyecegini belirttiginizde, gercekten degismeyecegini artik garanti eder. Bu nedenle o degeri artik kendiniz takip etmek zorunda kalmazsiniz. Sonuc olarak programiniz icin belirlediginiz prototip icin akil yurutmek daha kolay olacaktir.

Ancak degisilebilirlik (mutability) bazi durumlarda cok faydali olabilir ve programiniz icin kod yazmayi daha uygun hale getirebilir. Degiskenler yalnizca varsayilan olarak degismezdir, eger degisken adinin onune **"mut"** eklerseniz artik degiskeniniz degisebilir bir degisken olarak programin icerisinde yasamina devam eder. Mut anahtarini eklemek kodun diger bolumlerinin bu degiskenin degerini degistirecegini belirterek kodun gelecekteki okuyucularina da iletir.

Simdi src dizini altindaki main.rs kaynak kodu dosyamizi degistirelim:

$ vi main.rs

```
fn main() {
    let mut x = 5;
    println!("x degiskenine atanan deger: {}", x);
    x = 6;
    println!("x degiskeninin yeni aldigi deger: {}", x);
}
```

Gerekli degisikligi yapip **“mut”** anahtarini ekledikten sonra Cargo aracini kullanarak tekrar calistiralim:

$ cargo run

```
  Compiling mutability v0.1.0 (/Users/zgr/Desktop/Projects/mutability)
   Finished dev [unoptimized + debuginfo] target(s) in 1.15s
    Running `target/debug/mutability`
x degiskeni icin atanan deger: 5
x degiskeninin aldigi yeni deger: 6
```

Goruldugu uzere **“mut”** anahtari kullanildiginda x degiskenine ilk baglanan deger 5 iken daha sonra 6 degeri atanarak bu atamaya izin verildi ve degiskenimiz artik yeni degeri tasimaya basladi. Hatalarin onlenmesinden ziyade bir baska konu ise takas yani odunlesimdir (tradeoff). Ornegin, buyuk veri yapilari kullanildiginda, bir degiskeni yerinde degistirmek, yeni tahsis edilen degiskenleri kopyalayarak geri dondurmekten hizli olabilir. Daha kucuk veri yapilarinda ise yeni ornekler olustgurmak ve daha islevsel bir programlama konseptinde yazmak bazen daha kolay olabilir, bu nedenle daha dusuk performans bu netligi elde etmek icin bazi durumlarda degerli bir ceza olabilir.

Sabitler:

Bir onceki konuda bahsedilmis oldugu uzere degismek degiskenler gibi sabitler de (constants) bir ada bagli ve degismesine izin verilmeyen degerler alirlar. Ancak sabitler ve degiskenler arasinda bazi farklar mevcuttur.

Ilk olarak, “mut” anahtarinin sabitlerle kullanilmasina izin verilmez. Ayrica sabitler yalnizca varsayilan olarak degismez degildir kisaca her zaman degismezdirler. Sabitleri ise **“let”** anahtar sozcugu yerine **“const”** anahtar sozcugunu kullanarak bildirirsiniz ve degerin turu mutlaka aciklama icermelidir. Bir sonraki “Veri Turleri” bolumunde turleri ve bu turlerin ek aciklamalarini ele almak uzereyiz, bu nedenle su anda ayrintilar icin pek endise etmenize gerek yok. Her zaman ture aciklama eklemeniz gerektigini bilmeniz simdilik yeterli olacak.

Sabitler, global kapsam da dahil olmak uzere herhangi bir kapsamda bildirilebilir, bu da onlari kodun bircok bolumunun bilmesi gereken degerler icin faydali kilar. Son fark, sabitlerin yalnizca calisma zamaninda hesaplanabilecek bir degerin sonucu olarak degil, yalnizca sabit bir ifadeye ayarlanabilmesidir.

Simdi bir sabit bildirelim:

```
const 3_SAATIN_SANIYELERI u32 = 60 * 60 * 3;
```

Burada sabitimizin ismi 3_SAATIN_SANIYELERI'dir ve degeri 60 (bir dakikadaki saniye sayisi) ile 60 (bir saatteki dakika sayisi) ile 3 (program icerisinde saymak istedigimiz toplam sure saat cinsinden) carpilmasinin sonucunu verecektir. Rust'in sabitler icin adlandirma kurali, sozcukler arasinda alt cizgi ile tumunde buyuk harf kullanilmasidir. Derleyici, derleme zamaninda sinirli bir dizi islemi degerlendirir; bu, bu sabiti 10.800 degerine ayarlamak yerine, bu degeri anlasilmasi ve dogrulanmasi daha kolay bir sekilde yazmayi secmemize olanak tanir. Sabitleri bildirirken hangi islemlerin kullanilabilecegi hakkinda daha fazla bilgi icin Rust Referans'in sabit degerlendirme bolumune goz atabilirsiniz.

Bu sekilde sabitler programinizda yasadigi tum dogal sure boyunca, bildirdikleri kapsam dahilinde gecerlidir. Bu ozellik, sabitleri, uygulama etki alaninizdaki herhangi bir maksimum nokta sayisi gibi, programin birden fazla bolumunun bilmesi gerekebilecek degerler icin kullanisli hale getirir. Kisaca bir oyunun oyuncusunun isik hizinda kazanmasina izin verilir.

Programiniz boyunca kullanilan sabit kodlanmis degerleri sabitler olarak adlandirmak, bu degerin anlamini kodun gelecekteki koruyucularina iletmede faydalidir. Ayrica, gelecekte sabit kodlanmis degerin guncellenmesi gerekiyorsa degistirmeniz gereken kodunuzda yalnizca bir yere sahip olmaniza da yardimci olur.

Golgeleme:

Bir sonraki bolumde yer alacak tahmin oyunu egitiminde goreceginiz gibi onceki degiskenle ayni ada sahip yeni bir degisken bildirebilirsiniz. Rustacean'lar birinci degiskenin ikinci tarafindan golgelendigini (shadowing) soylerler; bu, ikinci degiskenin degerinin, degisken kullanildiginda programin gordugu sey oldugu anlamina gelmektedir. Ayni degiskenin adini kullanarak ve **"let"** anahtarini tekrarlayarak bir degiskeni golgelemek mumkundur.

Simdi bunu bir ornekle gorelim, kaynak kodunuza gidin ve asagidaki kodu kopyalayin.

$ cat src/main.rs

```
fn main() {
    let x = 5;
    let x = x + 1;
    {
        let x = x * 2;
        println!("x degiskeninin ic kapsamda degeri: {}", x);
    }
    println!("x degiskeninin yeni degeri: {}", x);
}
```

$ cargo run

```
   Compiling mutability v0.1.0 (/Users/zgr/Desktop/Projects/shadowing)
    Finished dev [unoptimized + debuginfo] target(s) in 0.99s
     Running `target/debug/mutability`
x degiskeninin ic kapsamda degeri: 12
x degiskeninin yeni degeri: 6
```

Programimiz once x degiskeni icin 5 degerini atar. Ardindan let x = tekrar edilerek (kopyalanarak) ve orijinal degeri alinarak ve uzerine kodda da yazildigi uzere +1 degeri eklenerek golgelenir, boylece x degiskeninin degeri 6 olmustur. Daha sonra bir ic kapsamda ucuncu bir let anahtari da golgelenir, x degiskeni onceki degeri 2 ile carpilacak sekilde guncellenir ve 12 degerini alir. Bu ic kapsam sonlandiginda golgeleme (shadowing) sonra erer ve x 6 degerine tekrar doner. Programi calistirdiktan sonraki sonuclarina baktiginizda once ic kapsam ({} blogu icerisinde yer alan) islemin yapilarak sonuclandigini daha sonra ise ana fonksiyon kapsaminda ayni degiskenin son degerine dondugunu goreceksiniz.

Son olarak golgeleme bir degiskenin **“mut”** olarak isaretlenmesinden farklidir cunku **“let”** anahtari kullanmadan yanlislikla bu degiskene yeniden atama yapmaya calisirsak derleme zamaninda hata aliriz. Ancak **“let”** kullanarak birkac donusum seklinde deger almasini saglayabiliriz ve en sonunda degiskenin degismez (immutability) olmasini saglayabiliriz.

Ayni zamanda **“mut”** ve golgeleme arasindaki baska bir fark, **“let”** anahtarini tekrar kullandigimizda etkin bir sekilde yeni bir degisken olusturdugumuz icin degerin turunu degistirebilir ancak ayni adi yeniden kullanabiliriz. Ornekleyecek olursak, programimizin bir kullanicidan bosluk karakterleri girerek bazi metinler arasinda kac bosluk istedigini gostermesini istedigini ve ardindan bu girisi bir sayi olarak saklamak istedigimizi varsayalim:

```
let spaces = “     “;
let spaces = spaces.len();
```

Ilk bosluk degiskeni bir dize turudur ve ikinci bosluk degiskeni bir sayi turudur. Boylece golgeleme, bizi space_str ve space_num gibi iki farkli isimler bulmaktan kurtarir. Bunun yerine daha basit bosluk adini yeniden kullanabiliriz. Ancak yukarida gosterildigi gibi bunun icin mut kullanmak istersek bir derleme zamani (compile time) hatasi aliriz:

```
let mut spaces = “     “;
spaces = spaces.len();
```

$ cargo run

```
 Compiling variables v0.1.0 (/Users/zgr/Projects/variables)
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`
For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` due to previous error
```

Artik degiskenlerin nasil calistigini isledigimize gore sahip olabilecekleri veri turlerine bir goz atma vakti gelmis demektir!

Veri Turleri (Tipleri):

Rust Programlama Dili'ndeki her deger, Rust'a ne tur verilerin belirtildigini soyleyen ve bu verilerle nasil calisilacagini bilen belirli veri turleridir (tipleridir). Bu noktada Rust icin iki veri turune bakiyor olacagiz; bunlar, Skaler (Sayisal) ve Bilesik (Scalar and Compound).

Rust statik olarak yazilmis bir dildir. Bu derleme zamaninda tum degiskenlerin turlerini bilmesi gerektigi anlamina gelir. Derleyici genellikle degere ve onu nasil kullandigimiza bagli olarak ne tur bir veri turu kullanmak istedigimizi cikarabilir. Bircok turun mumkun oldugu durumlarda, ornegin bir sonraki bolumde uzerinde duracagimiz oyun orneginde, ayristirma yontemini kullanarak bir diziyi nasil sayisal bir ture donusturdugumuzde, asagidaki gibi bir tur ek aciklamasi da eklemeliyiz:

```
let guess: u32 = “42”.parse().expect(“Bir numara degildi!”);
```

Yukaridaki ornekte goruldugu gibi bir tur aciklamasi eklenmez ise Rust asagidaki hatayi goruntuler; bu, derleyicinin hangi turu kullanmak istedigimizi bilmesi icin bizden daha fazla bilgiye ihtiyaci oldugu anlamina gelir:

$ cargo build

```
   Compiling no_type_annotations v0.1.0 (/Users/zgr/projects/no_type_annotations)
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Bir numara degildi!");
  |         ^^^^^ consider giving `guess` a type
For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

Diger veri turleri icin farkli tur aciklamalar goreceksiniz.

Skaler Tipler:

Bir skaler tip, tek bir degeri temsil eder. Rust'in dort tur birincil skaler turu vardir: tamsayilar, kayan noktali sayilar, boole'ler ve karakterler. (integers, floating point numbers, booleans, characters).

Tamsayilar (Integers):

Tamsayilar, kesirli bileseni olmayan sayilardir. Bu tur bir bildirim, atandigi degerin 32 bit yer kaplayan isaretsiz bir tamsayi (isaretli tamsayilar tur bildiriminde u yerine i ile baslarlar) olmasi gerektigini belirtir. Bir tamsayi degerinin turunu bildirmek icin bu degiskenlerden herhangi birini kullanabilirsiniz:

Uzunluk	Isaretli	Isaretsiz
8-bit	i8	u8
16 bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Her varyant imzali veya imzasiz daha dogrusu isaretli veya isaretsiz olabilirler ancak bir boyuta sahiptir. Degerin negatif olmasinin mumkun olup olmadigina bagli olarak baska bir deyisle sayinin onunla bir isareti olmasi gerekip gerekmedigi (imzali, isaretli) veya yalnizca pozitif olup olmayacagina ve bu nedenle isaretsiz olarak gosterilip gosterilmeyecegine (imzasiz, isaretsiz) atifta bulunur.

Bu tipki kagida sayilar yazmak gibidir; isaret onemli oldugunda, bir sayi arti veya eksi isaretiyle gosterilir; ancak, sayinin pozitif oldugunu varsaymak guvenli oldugunda isaretsiz yani imzasiz olarak gosterilir. Isaretli (signed) sayilar, ikinin tumleyen gosterimi kullanilarak saklanir.

Her isaretli (signed) varyant -(2n – 1) ile 2n - 1 – 1 arasindaki sayilari saklayabilir, burada n, varyantin kullandigi bit sayisidir. Boylece i8 ornegin -(27) ile 27 – 1 arasindaki sayilari saklayabilir, bu da -128 ila 127’ye esittir. Isaretsiz degiskenler 0 ila 2n – 1 arasindaki sayilari saklayabilir, dolayisi ile bir u8 0 ila 28 – 1 arasindaki sayilari saklayabilir bu da 0 ila 255’e esittir.

Tamsayi degismezlerini tabloda yer alan herhangi bir bicimde yazabilirsiniz. Birden cok sayisal tur olabilen sayi degismezlerinin 57u8 gibi bir tur son ekinin tur belirlemesine izin verdigini aklinizda tutun. Sayi degismezleri, ornegin deger belirtilerek 1000 olacagi gibi gorsel bir ayirici (_) kullanilarak ornegin 1_000 seklinde de kullanilabilirler.

Peki hangi tamsayi turunu kullanacaginizi nasil bileceksiniz?

Emin degilseniz, Rust'in varsayilanlari genellikle baslamak icin iyi bir yaklasimdir: tamsayi turleri varsayilan olarak i32 turundedir. Isize veya usize kullandiginiz durumlar genellikle bir tur dizgisine degisken eklerken gerceklesir.

Desimal (Onluk)	98_222
Heks (Onaltilik)	0xFF
Oktal (Sekizlik)	0o77
Ikilik	0b1111_0000
Bayt	b'A'

Kayan Noktali Tipler (Floating Point Types):

Rust ayrica ondalik basamakli sayilar olan kayan noktali sayilar icin iki ilkel tur barindirir. Rust'in bu kayan nokta turleri sirasi ile 32-bit ve 64-bit boyutunda olan f32 ve f64 turleridir. Varsayilan tur f64 turudur cunku moden islemcilerde (CPU) kabaca f32 ile ayni hizdadir ancak daha fazla hassasiyete sahiptir. Tum kayan noktali tipler (floating point types) imzali (signed) olarak isaretlenmistir.

Iste bir ornek:

```
fn main() {
        let x = 2.0; // varsayilan olarak f64
        let y: f32 = 3.0; // f32 32-bit olarak
}
```

Kayan noktali sayi tipleri IEEE-754 standardina gore Rust icerisinde temsil edilmistir ve f32 turu tek hassasiyetli (single precision) bir kayan noktadir ancak f64 cift hassasiyetlidir (double precision).

Numerik Operatorler (Numeric Operations):

Rust tum sayi turleri icin beklediginiz temel aritmetik islemleri destekler. Bu sayede toplama, cikarma, bolme ve kalan hesaplama gibi islemleri varsayilan olarak yapabilirsiniz.

Ayrica bir tamsayiyi (integer) bolme islemi aslinda en yakin tam sayiya yuvarlamaktir. Asagidaki kod, bir **let** ifadesinde her bir sayisal islemi nasil kullanacaginizi ornekler:

```
fn main() {
    // ekleme
    let sum = 5 + 10;

    // cikarma
    let difference = 95.5 - 4.3;

    // carpma
    let product = 4 * 30;

    // bolme
    let quotient = 56.7 / 32.2;
    let floored = 2 / 3; // Sonuc 0 olacaktir.

    // kalana yuvarlama
    let remainder = 43 % 5;
}
```

Yukarida gordugunuz tum ifadelerde her ifade bir aritmetik operator kullanir ve daha sonra bir degiskene baglanan tek bir deger olarak sonlanir. “EK B” iceriginde Rust’in tum operatorlerinin bir listesini bulabilirsiniz.

Boole Tipleri (Boolean Type):

Daha once bir programlama dili kullandi iseniz ya da programlamaya asina iseniz Boole (boolean) tiplerinin de ayni mantikta Rust icinde calistigindan emin olabilirsiniz. Buna gore bir Boole turu iki olasi deger alirlar ve bunlar **“true”** ya da **“false”** degerleridir. Tum boole turlerinin boyutu bir bayttir ve Rust icerisinde boole turleri **“bool”** anahtar kelimesi kullanilarak derleyiciye belirtilir.

Hemen bir ornek uzerinden bunu gorelim:

```
fn main() {
        let t = true;
        let f: bool = false // tur icin aciklama eklendi (annotation)
}
```

Boole turlerinden donen degerleri kullanmanin ana yolu, “**if”** ifadesi gibi kosullu ifadelerdir. Rust’a ifadelerin nasil calistigina ilerideki bolumlerde **“Kontrol Akisi”** bolumunde yer verecegiz.

Karakter Tipleri (Character Types):

Rust programlama dili icerisinde yasayan karakter tipleri (character types) en ilkel alfabetik turdur. Iste **“char”** karakter turu kullanilarak bildirilen degerlere bazi ornekler:

```
fn main() {
        let k = ‘z’;
        let z = ‘Z’;
        let kalp_gozlu_kedi = '  ';
}
```

Cift tirnak (“ ”) kullanan dize degismezlerinin aksine, tek tirnakli (’’) karakter degismezlerini belirtiyor oldugumuzu gozden kacirmayin. Rust’in **“char”** turu, dort bayt boyutundadir ve bir **Unikod Skaler Deger**’ini temsil eder. Buna gore ASCII karakter setinden cok daha fazlasinin temsil edilebilecegini aklinizda bulundurun.

Bilesik Tipler (Compound Types):

Bilesik turler, birden cok degeri tek bir turde gruplamaktir. Rust’in iki ilkel bilesik turu vardir: demetler ve diziler (tuples and arrays).

Demetler (Tuple Types):

Demetler, cesitli turlere sahip bir dizi degeri tek bir bilesik turde gruplandirmanin genel bir yoludur. Demetler sabir bir uzunluga sahiptir: bir kez bildirirseniz bundan sonra buyutemez ve kucultemezsiniz.

Parantez icinde virgulle ayrilmis bir degerler listesi yazarak bir demet olusturabilirsiniz. Tanimlama grubundaki her konumun bir turu vardir ve tanimlama grubundaki farkli degerlerin turlerinin de ayni olmasi gerekmez. Bu ornekte istege bagli tur ek aciklamalarina yer verdik:

```
fn main() {
        let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

Ornekte gordugunuz gibi degisik tipteki degerler demete (tuple) baglanirlar, cunku bir demet tek bir bilesik eleman olarak kabul gorecektir. Bir tanimlama grubundan tek tek degerleri elde etmek icin, bir tanimlama grubunu yok etmek icin desen eslestirme yontemini (pattern matching) kullanabiliriz:

```
fn main() {
        let tup = (500, 6.4, 1);
        let (x, y, z) = tup;
        println!("Demet icindeki degerler: {}, y);
}
```

Bu program once bir tanimlama grubu olusturur ve onu **"tup"** degiskenine atar. Daha sonra tekrar **"let"** tanimlanarak x, y ve z gibi uc ayri degiskene uc ayri deger atanir. Bu modele eslestirme yontemi (pattern matching) denir. Bu ayni zamanda bir yikimdir (destructuring) cunku bir demeti uc parcaya parcalidiniz. Son satirda ise cikti olarak y degiskeninin degerini yani 6.4 degerini ekrana yazdiracaktir.

Ayrica, nokta (.) kullanarak demet icerisindeki degerin indisini belirterek grup ogelerine dogrudan erisebilirsiniz.

```
fn main() {
        let x: (i32, f64, u8) = (500, 6.4, 1);
        let besyuz = x.0
        println!("io32 tipindeki x degiskenin ilk degeri: {}, besyuz);
}
```

Dizi Turleri (Array Types):

Birden cok degerden olusan bir koleksiyona sahip olmanin baska bir yolu da dizileri (array) kullanmaktir. Demetlerden (tuple) farkli olarak dizilerde yer alan her elemanin tipi ayni olmalidir. Diger bazi dillerdeki yaklasimin aksine Rust'in dizilerinin sabit bir uzunlugu oldugunu unutmayin.

Bir dizideki degerlerio koseli parantezler icinde virgulle ayrilmis bir liste olarak yazdiralim:

```
fn main() {
        let a = [1, 2, 3, 4, 5];
}
```

Diziler, verilerinizin bir yigin icerisinde olmasi yerine yigina tamamen atanmasini istediginizde veya her zaman sabit sayida ogeye sahip oldugunuzdan emin olmak istediginizde oldukca yararlidir. Yine de bir dizi, vektor turu kadar esnek degildir. Vektor, standart kitaplik tarafindan saglanan ve boyutunun buyumesine veya kuculmesine izin verilen benzer bir turdur. Dizi mi yoksa vektor mu kullanacagizdan emin olmadiginiz durumlarda yuksek ihtimalle bir vektor kullanmalisiniz. Bu nedenle vektorleri acikladigimiz bolumu dikkatli incelemelisiniz.

Ayni zamanda diziler eleman sayisinin degismesi gerekmedigini bildiginiz zaman daha kullanislidir. Ornegin, bir programda ayin adlarini kullaniyor olsaydiniz her zaman 12 oge icerecegini bildiginiz icin vektor yerine dizi kullanirdiniz:

```
        let aylar = [“Ocak”, “Subat”, “Mart”, “Nisan”, “Mayis”, “Haziran”, “Temmuz”, “Agustos”, “Eylul”, “Ekim”, “Kasim”, “Aralik”]
```

Dizinin Ogelerine Erisim:

Dizin, yigin halinde ve boyutu bilinen sabit boyuttaki tek bir bellek yiginidir. Ancak yine de bir dizinin ogelerine asagidaki gibi erisebilirsiniz:

```
fn main() {
        let a = [1, 2, 3, 4, 5];
        let ilk = a[0];
}
```

Gecersiz Dizi Ogelerine Erisim:

Dizinin sonunu asan bir dizinin ogesine erismeye calisirsaniz ne olacagini gorelim. Kullanicidan bir dizi indisi almak icin tahmin oyunu orneginde oldugu gibi asagidaki kodu calistirdiginizi varsayalim:

```
use std::io;

fn main() {
    let dizi = [1, 2, 3, 4, 5];

    println!("Lutfen erismek istediginiz ogenin indisini girin: ");
    let mut indis = String::new();

    io::stdin()
        .read_line(&mut indis)
        .expect("Satir okunamadi.");

    let indis: usize = index
        .trim()
        .parse()
        .expect("Girdiginiz indis bir sayi degildi.");

    let eleman = dizi[indis];
    println!(
        "Erismek istediginiz {} ogesinin degeri: {}",
        indis, eleman
    );
}
```

Yukaridaki kod basari ile derlendiginde ve calistirdiginizda diziye ait olan indislerden herhangi birini girdi olarak belirtirseniz program size o dizideki indise karsilik gelen degiskenin degerini dondurur. Ancak indis olmayan ornegin 10 gibi bir girdi verirseniz Rust size dizinin toplam uzunlugunu ve erismeye calistiginiz indisin olmadigini iceren bir hata donecektir.

Islevsellik (Functions):

Fonksiyon (islevsellik ya da ingilizce tanimi ile functions) kullanimi Rust programlama dilinde oldukca yaygindir. Dildeki en onemli islevlerden birini zaten gordunuz: bircok programin giris noktasi olan ana fonksiyon (func main). Ayrica, yeni islevler bildirmenizi saglayan **“fn”** anahtarini kullanarak yeni fonksiyonlar da olusturabileceginizi gordunuz.

Rust kodu, tum harflerin kucuk oldugu ve ayri sozcukler kullanilmis ise arada cizgi ile ayrim oldugu, islev ve degisken adlari icin geleneksel stil olarak yilan stilini (snake case) kullanir. Ornek bir fonksiyon tanimini iceren program soyledir:

```
func main() {
        println!(“Merhaba, Dunya!”);
        baska_bir_fonksiyon();
}

fn baska_bir_fonksiyon() {
        println!(“Bu da baska bir fonksiyon ancak size ana fonksiyon ile dondu.”);
}
```

Rust dilinde fonksiyon tanimlarken **fn** anahtari ve bir islev ismi belirleriz ve bir dizi parantez “()” girerek islemi tamamlariz. Kivrimli parantezler “{}” ise derleyiciye bir fonksiyonun nerede baslayip nerede bittigini bildirir.

Tanimladiginiz herhangi bir fonksiyonu (islevsellik) adini ve ardindan yine bir dizi parantez “()” girerek cagirabilirsiniz. Programinizda baska_bir_fonksiyon_daha gibi bir islev tanimli oldugunda yine ana fonksiyon (main func) icinden cagirmaniz mumkundur. Bunun sirasi pek onemli degildir cunku Rust fonksiyonlarinizi nerede tanimladiginizla ilgilenmez, yalnizca bir yerde tanimlanmalari yeterlidir.

$ cargo run

Merhaba, Dunya!

Bu da baska bir fonksiyon ancak size ana fonksiyonla dondu.

Parametreler (Parameters):

Rust dilinde fonksiyonlarimizi, bir fonksiyonun iceriksel parcasi olacak sekilde ozel degiskenler olan parametrelere sahip olacak sekilde tanimlayabiliriz. Bir islevin parametreleri oldugunda, ona bu parametreler icin somut degerler saglayabilirsiniz. Teknik olarak somut degerlere arguman (argument) adi verilir. Gundelik konusmalarda insanlar parametre ve arguman kelimelerini bir fonksiyonun tanimindaki degiskenler veya bir fonksiyonu cagirdiginizda iletilen somut degerler icin birbirinin yerine kullanma egilimindedir.

Bununla ilgili hemen bir ornek yapalim ve yukaridaki ornegimizde varolan ikincil fonksiyonumuza parametreler vererek bazi degerleri ya da tanimlari arguman olarak anlamasini saglamis olalim.

```
fn main() {
        ikincil_fonksiyon(5);
}

fn ikincil_fonksiyon(x: i32) {
        println!(“ikincil fonksiyonda verilen x degiskeninin ana fonksiyonda aldigi deger {}”, x);
```

Bu programi calistirdigimizda elde edecegimiz sonuc asagidaki gibi olacaktir.

$ cargo run

ikincil fonksiyonda verilen x degiskeninin ana fonksiyonda aldigi deger: 5

Burada **ikincil_fonksiyon** bildirimi, x adinda bir parametre almis ve x degiskeninin tipi **i32** olarak belirtilmistir. 5 sonucu ise ana fonksiyonda **ikincil_fonksiyon**’a parametre olarak atandigi icin **println!** makrosu ana fonksiyonu islettiginde x icin 5 degerini donecektir. Fonksiyon imzalari (function signatures) her parametrenin de turunu bildirmenizi ister. Bu Rust’in tasariminda bilincli olarak verilmis bir karardir.

Ifadeler ve Deyimler (Statements and Expressions):

Fonksiyon govdeleri, istege bali olarak bir ifadeyle biten bir dizi ifadeden olusur. Rust ifade tabanli (statements) bir dil oldugundan, bu anlasilmasi gereken onemli bir ayrimdir. Diger diller ayni ayrimlara sahip degildir. O halde simdi hangi ifadelerin ve deyimlerin olduguna ve farkliliklari uzerinde durarak islevleri yani fonksiyonlari nasil etkiledigini gozlemleyelim.

Aslinda zaten ifadeleri ve deyimleri kullandik. “let” anahtari ile bir degisken olusturmak ve ona bir deger atamak ifadedir (statement). Yani y = 6 gibi bir ornekte bu bir ifadedir.

```
fn main() {
        let y = 6;
}
```

Yukarida da gordugunuz uzere islev tanimlamalari da birer ifadelerdir, ve yukaridaki ornek ana islev (main func) dahil olmak uzere kendi icinde bir ifade icerir. Ifadeler deger dondurmezler. Bu nedenle, asagidaki gibi bir kod yazarsaniz yani baska bir degiskene let ifadesi atarsaniz hata alirsiniz:

```
fn main() {
        let x = (let y = 6);
}
```

$ cargo run

note: variable declaration using `let` is a statement

Cunku let y = 6 ifadesi bir deger dondurmez, dolayisi ile x degiskeninin baglanabilecegi bir deger yoktur. Bu, atanan degeri donduren C veya Ruby gibi dillerden farklidir. Bu dillerde x = y= 6 gibi bir ifade kullanarak hem x hem de y icin 6 degerini atayabilirsiniz. Rust icin bu durum gecerli degildir.

Ancak deyimler (expressions) ifadeler gibi degildir ve Rust derleyicisi tarafindan bir deger olarak degerlendirilir ve Rust’ta yazacaginiz kodun geri kalaninin cogunu olusturur. 10 degerini veren matematiksel bir aritmetik ifade olan 5 + 5 islemini dusunun. Bu durumda ilk 5 ikinci 5 sonucunu donen bir deyimdir ve bunu cagiran islevsellik de bir deyimdir. Rust programlama dilinde makro kullanimi da bir deyimdir. Kivrimli parantezler “{}” kullanilarak olusturulan kapsam blogu da bir deyimdir ve bu deyimleri kullanarak 5 + 5 = 10 aritmetigini kodlayabilirsiniz.

```
fn main() {
        let a = { let b = 5; b + 5 };
        println!("5 + 5 toplaminin sonucu: {}", a);
}
```

$ cargo run

5 + 5 toplaminin sonucu: 10

Donus Degerleri Olan Fonksiyonlar (Functions with Return Values):

Rust Programlama Dili'nde fonksiyonlar, onlari cagiran koda degerler dondurebilir. Donus degerlerini adlandirmiyoruz, ancak turlerini bir ok isareti ile (->) bildirebiliyoruz. Rust'ta islevin donus degeri, bir islevin govdesinin blogundaki son ifadenin degeri ile es anlamlidir. Ayni zamanda return anahtarini kullanarak ve bir deger belirterek bir islevden daha erken bir donus (return) saglayabilirsiniz, ancak cogu islev son ifadeyi ortuk olarak dondurecektir. Asagida deger donduren bir fonksiyon ornegine goz atalim:

```
fn kirkiki() -> i32 {
   42
}
fn main() {
   let a = kirkiki();
   println!("Hayatin anlami: {}", a);
}
```

Yukaridaki kirkiki fonksiyonunda hicbir fonksiyon cagrisi, makro hatta "let" deyimi dahi yoktur, tek basina 42 sayi degerini goruyorsunuz. Bu, Rust'ta tamamen gecerli bir islevselliktir. Fonksiyonun donus turunun "i32" olarak belirlendigine dikkat edin. Simdi bu kodu calistiralim:

$ cargo run

Hayatin anlami: 42

42 kirkiki fonksiyonunun donus degeridir, bu nedenle donus turu de i32 olarak belirtildi. Bunu daha detayli inceleyecek olursak iki onemli bit ile karsilasiriz. Birincisi let a = kirkiki(); bir degiskeni baslatmak icin bir fonksiyonun donus degeri kullandigimizi gosterir. Ikincisi ise kirkiki fonksiyonunun goruldugu uzere parametresi yoktur ve sadece donus degerinin turu tanimlanmistir. Ancak islevin govdesi, degerini dondurmek istedigimiz bir deyim oldugu icin noktali virgul (;) icermez.

Baska bir ornege daha goz atalim.

```
fn main() {
        let x = bes(5);
        println!(“x degiskeninin degeri: {}”, x);
}

fn bes(x: i32) -> i32 {
        x + 1
}
```

Yukaridaki kod bir noktada dogrudur yani x once i32 tipinde bir 5 degeri almis ve sonra +1 degere atanarak sonucun donus degeri alan ana fonksiyonla ekrana iletilmesi saglanmistir. Ancak x + 1 deyimine noktali virgul (;) eklersek kodumuz hata verecektir:

```
fn bes(x: i32) -> i32 {
        x + 1;
}
```

$ cargo run

error[E0308]: mismatched types

Peki bu neden oldu? Cunku biz bes fonksiyonumuzda donus tipini i32 olarak belirtmistik ve ana fonksiyonumuza x + 1; bir ifade olarak verildiginde tip hatasi aldik. Bu gibi durumlarda Rust derleyicisi yardimci mesaj saglar ve noktali virgulun (;) kaldirilmasini onerir.

Yorum Satirlari (Comments):

Butun programlama dillerinde genellikle oldugu uzere kodlarin anlasilmasini kolaylastirmak icin caba gosterilir ve bazen ek aciklamalara yer verilir. Bu durumlarda programcilar bu aciklamalari belirtmek icin derleyicinin gormezden gelecegi yorum satirlari isaretlerler, bu yorum satirlari derleyici tarafindan es gecilir ancak kaynak koduna goz atan insanlar icin faydali olurlar. Rust dilinde bu iki egik cizgi isareti (“//”, two slashes) ile saglanir. Bir ornek verecek olursak:

```
// merhaba, dunya!
```

Akis Kontrolleri (Control Flow):

Bir kosulun dogru olup olmadigina dayali bazi kodlari calistirma veya bir kosul dogruyken bazi kodlari tekrar tekrar calistirma yetenegi, cogu programlama dilinde temel yapi taslaridir. Rust kodunun yurutme akisini kontrol (control flow) etmeninizi saglayan en yaygin yapilar **“if”** deyimleri (edxpressions) ve dongulerdir (loops).

if Deyimleri (if expressions):

Bir if deyimi, kosullara bagli olarak kodunuzu dallandirmaniza olanak tanir. Bir kosulu saglarsaniz ve ardindan “bu kosul karsilanirsa, bu kod blogunu calistir. Kosul karsilanmaz ise bu kod blogunu calistirma.” seklinde bir akis kontrolune sahip olursunuz.

Simdi bunu kesfetmek icin akis adinda bir proje yaratalim ve cargo aracinin olusturdugu src/main.rs kaynak kodu dosyasini asagidaki gibi duzenleyelim:

```
fn main() {
        let numara = 3;
if number < 5 {
        println!("kosul: true");
} else {
        println!("kosul: false");
 }
}
```

Yukaridaki ornekte de oldugu uzere tum if deyimleri **“if”** anahtar kelimesi ile baslar ve ardindan bir kosul icerir. Boyle bir durumda, degisken sayisinin 5’ten kucuk bir degere sahip olup olmadigi kontrol edilir. Kosul dogruysa yurutulecek kod blogunu kosulun hemen ardindan kume parantezleri icine yerlestiririz. Kosulun yanlis olarak degerlendirildigi blok **“else”** anahtari ile belirtilir ancak bu istege baglidir. Alternatif bir yanlis degerlendirme esnasinda programin nasil akacagini bu sekilde belirtebiliriz.

$ cargo run

kosul: true

Gordugunuz gibi numara degiskeninin aldigi deger 5 sayisindan kucuk oldugu icin programimiz **“true”** olarak sonuc dondu. Eger numara degiskenine (variable) ornegin 7 sayisini atayacak olsaydik programimiz bir sonraki else kosuluna uygun olarak “false” kondisyonunda sonuc donecekti.

```
    let numara = 7;
```

$ cargo run

kosul: false

Bu arada yukaridaki if deyimi kosulunda bir boole (boolean) ifadesi kullandigimizi da goz onunde bulundurun. Kosul bir boole olmasaydi hata alirdiniz. Hemen bir ornek daha yapalim:

```
fn main() {
    let numara = 3;
if number {
    println!("numara degiskenimiz uc");
 }
}
```

 # cargo run

error[E0308]: mismatched types

Bu hata mesajini daha once de gordunuz. Rust’in bir boole bekledigini ancak bir tamsayi aldigini gosterir. Ruby veya JavaScript gibi dillerin aksine Rust, boolean olmayan turleri otomatik olarak boolean’a donusturmeye calismaz. Rust ile kod yazarken yeterince acik olmalisiniz ve her zaman kosullu ifadelerde bir bir boolean olup olmadigini bildirmelisiniz. Ancak bazi durumlarda karsilastirma operatoru kullanilirken bu gerekmez.

Simdi bir ornekle bunu aciklayalim:

```
fn main() {
        let numara = 3;
if number !=0 {
        println!("Sifirdan farkli bir deger atadiniz.");
 }
}
```

Yukaridaki ornekte programiniz cikti olarak “Sifirdan farkli bir deger atadiniz” seklinde mesaj vererek sonlanacaktir. Burada bir karsilastirma operatoru (=) ancak “!=” seklinde kullandigimiza dikkat edin.

else if ile Birden Cok Kosulun Akis Kontrolu (Handling multiple conditions with else if):

Bir **“else if”** deyimi ile akis saglayarak programinizda **if** ve **else**’leri ic ice kullanarak birden cok kosul kullanabilirsiniz. Hemen bir ornek yapalim:

```
fn main() {
   let numara = 6;
   if numara % 4 == 0 {
      println!("Sayi 4 ile tam bolunebilir.");
   } else if numara % 3 == 0 {
      println!("Sayi 3 ile tam bolunebilir.");
   } else if numara % 2 == 0 {
      println!("Sayi 2 ile tam bolunebilir.");
   } else {
      println!("Sayi 2,3 ve 4 ile tam bolunemez.");
   }
}
```

Yukaridaki programin alabilecegi dort olasilik vardir. Bu programi yuruttugunuzde sirasi ile her bir **if** ifadesini kontrol edecek ancak kosulun dogru oldugu ilk blogu yurutecektir.

Ayrica 6 sayisi 2 sayisina tam bolunebilmesine ragmen, cikti olarak 2'ye bolunebildigini gormuyoruz ve else blogundan sayinin 2, 3, 4 sayilarina bolunemedigini gormuyoruz. Bunun nedeni Rust derleyicisinin blogu yalnizca ilk gercek kosul icin calistirmasi ve bir kez buldugunda gerisini kontrol etmemesidir.

Cok fazla **"else if"** deyimi kullanmak kodlama stilinizi karmasik hale getirebilir, bu nedenle birden fazla varsa kodunuzu yeniden duzenlemek (refactoring) zorunda kalacaksiniz. Bu durumlar icin **"Construct"** (eslesme) adini verdigimiz kod yapilarini kullanabilirsiniz. Bu ilerleyen bolumlerde detayli bir sekilde ele alinacaktir.

let deyimlerinde if kullanma (Using if in a let statement):

if bir deyim (expression) oldugu icin sonucu bir degiskene atamak icin **"let"** anahtarinin sag tarafinda kullanabiliriz:

```
fn main() {
   let kondusyon = true;
   let numara = if condition { 4 } else { 2 };

   println!("Alinan deger: {}", numara);
}
# cargo run
Alinan deger: 4
```

Rust'in kod bloklarinin iclerindeki son deyimleri degerlendirdigini ve sayilarin kendi baslarina da birer ifade oldugunu unutmayin. Bu durumda, **if** ifadesinin tamaminin degeri, hangi kod blogunun yurutuldugune baglidir. Bu ayni zamanda **if** deyiminden alinacak sonucun potansiyeline sahip degerlerin ayni tip olmasi gerektigi anlamina da gelir. Eger turler uyumsuz ise bir hata alacaksiniz. Bir ornekle inceleyelim:

```
fn main() {
   let kondusyon = true;
   let numara = if condition { 4 } else { "alti" };
   println!("Alinan deger: {}", numara);
}
```

Yukaridaki sekilde duzenledigimiz koda sahip programimizi calistiracak olursak asagidaki gibi bir hata aliriz.

 # cargo run

error[E0308]: `if` and `else` have incompatible types

Donguler (Loops):

Bir kod blogunu bir kereden fazla yurutmek genellikle yararlidir. Bu gorev icin Rust, dongu govdesi icindeki kodu sonuna kadar calistiracak ve ardindan hemen bastan baslayacak birkac dongu (loop) saglar. Donguleri denemek icin donguler adinda yeni bir proje yapalim.

Rust uc tur donguye sahiptir: **loop**, **while** ve **for**

Donguler Rust'a bir kod blogunu sonsuza kadar veya acikca durmasını soyleyene kadar tekrar tekrar yurutmesini iletir.

```
fn main() {
        loop {
                println!("tekrarliyoruz!");
 }
}
```

Yukaridaki koda sahip programimizi calistirdigimizda programimiz biz tekrar durdurana dek surekli olarak cikti uretecektir. Eger boyle bir donguyu test ediyor ve kesme (interrupt) gondermek istiyorsaniz control + c tuslarina basabilirsiniz.

$ cargo run

tekrarliyoruz!

Tekrarliyoruz!

^C

Neyse ki Rust kod kullanarak bir donguden cikmanin bir yolunu da sunar. Programa donguyu yurutmeyi ne zaman durduracagini soylemek icin **“break”** anahtarini dongunun icinde belirtebilirsiniz. Eger donguler icinde tekrar eden donguleriniz varsa, o noktada en icteki donguye uygulayin ve devam edin. Iste iki ic ice dongu iceren bir ornek:

```
fn main() {
        let mut counter = 0;
        'counter_up: loop {
        println!("counter = {}", counter);
        let mut remaining = 10;
        loop {
                println!("remaining = {}", remaining);
                if remaining == 9 {
                break;
                } if count == 2 {
                break 'counting_up;
        } remaining -= 1;
        } count += 1;
        } println!("End counter = {}", counter);
}
```

Yukaridaki ornegimiz dis dongude counting_up etiketine sahiptir ve 0’dan 2’ye kadar sayar. Etiketsiz ic dongu 10’dan 9’a geri sayim yapar. Bir etiket belirtmeyen ilk kesme (interrupt) yalnizca ic donguden cikar ve cointing_up ifadesi sonunda dis donguden cikar.

Yukaridaki programimizi calistirdigimizda asagidaki gibi bir cikti elde ederiz.

$ cargo run

counter = 0

remaining = 10

remaining = 9

counter = 1

remaining = 10

remaining = 9

counter = 2

remaining = 10

End counter = 2

Dongulerden Deger Dondurme (Returning Values from Loops):

Dongulerin bir baska kullanim amaci, bir is parcaciginin isini tamamlayip tamamlamadigini kontrol etmek gibi basarisiz olabilecegini dusundugunuz bir islemi yeniden denemektir. Ayrica, bu islemin sonucunu donguden kodugunuzun geri kalanina aktarmaniz gerekebilir. Bunu yapmak icin, donguyu durdurmak icin kullandiginiz "**break"** anahtarindan sonra dondurulmesini istediginiz degeri ekleyebilirsiniz; bu deger, burada gosterildigi gibi kullanabilmeniz icin donguden dondurulecektir:

```
fn main() {
   let mut counter = 0;
   let result = loop {
      counter += 1;
      if counter == 10 {
         break counter * 2;
      }
 }; println!("Sonuc {}", result);
}
```

Yukaridaki ornekte donguden once counter adinda bir degisken tanimliyoruz ve onu 0 degerinde baslatiyoruz. Ardindan donguden donen degeri tutacak result adinda bir degisken tanimliyoruz. Dongunun her yinelemesinde counter degiskenine 1 eklenir ve ardindan sayacin 10'a esit olup olmadigi kontrol edilir. Counter * 2 degerine ulastiginda **"break"** anahtari cagrilir ve bu durumda 20 olan result ekrana yazdirilir.

while ile Kosullu Donguler (Conditional Loops with while):

Bir programin genellikle bir dongu icindeki bir kosulu degerlendirmesi gerekebilir. Kosul dogru oldugunda dongu calisir. Kosul dogru olmadiginda, program **"break"** anahtarini cagirarak donguyu durdurur. Boyle bir davranisi dongu **if else** ve **break** kombinasyonunu kullanarak uygular. Buna uygun bir program yazmayi deneyebilirsiniz ancak bu model o kadar yaygindir ki, Rust, bunun icin **while** dongusu (while loop) adi verilen yerlesik bir yapiya zaten sahiptir.

```
fn main() {
   let mut sayi = 3;
   while sayi != 0 {
      println!("{}!", sayi);
      number -= 1;
   }
   println!("Vakit tamam!");
}
```

Bu yapi, dongu, if, else ve break kullandiysaniz gerekli olacak bircok ic ice yerlestirmeyi ortadan kaldirir ve bu daha aciktir. Bir kosul dogru oldugunda kod calisir; aksi durumda donguden cikacaktir.

for anahtari ile Dongu Kondusyonlari (Looping Condition with for):

Dizi gibi bir koleysiyonun ogeleri uzerinde dongu olusturmak icin while yapisini kullanmayi secebilirsiniz. Asagidaki ornek kod, dizideki tum ogeleri sayacaktir. 0 dizininde baslar ve dizideki son elemana ulasana kadar dongu devam edecektir.

```
fn main() {
   let a = [10, 20, 30, 40, 50];
   let mut index = 0;
   while index < 5 {
      println!("Deger: {}", a[index]);
      index += 1;
   }
}
```

$ cargo run

Deger: 10

Deger: 20

Deger: 30

Deger: 40

Deger: 50

Dizi degerlerinin tumu beklendigi gibi ciktida listelenecektir. Dizin bir noktada 5 degerine ulasacak olsa da, diziden altinci bir deger getirmeye calismadan once dongu yurutmeyi durdurur. Ancak bu yaklasim hataya aciktir; index degeri veya test kosulu yanlissa programin panige kapilmasina neden olabiliriz. Ornegin, a dizisinin tanimini dort ogeye sahip olacak sekilde degistirdiyseniz ancak index < 4 iken kosulu guncellemeyi unuttuysaniz, kod panikleyecektir. Ayrica yavastir, cunku derleyici dongu boyunca her yinelemede dizinin sinirlari icinde olup olmadiginin kosullu kontrolunu gerceklestirmek icin calisma zamaninda kodu ekler.

Bunun yerine bir alternatif olarak, bir **“for”** dongusu kullanabilir ve bir koleksiyondaki her oge icin bir miktar kod calistirabilirsiniz. Simdi bir ornek daha yapalim:

```
fn main() {
   let a = [10, 20, 30, 40, 50];
   for element in a {
      println!("the value is: {}", element);
   }
}
```

Bu kodu calistirdiginizda, bir onceki ornek kodla ayni ciktiya sahip oldugunu gorecegiz. Daha da onemlisi, artik kodun guvenligini artirdik ve dizinin sonunun otesine gecmek veya yeterince uzaga gitmemek ve bazi ogeleri kacirmaktan kaynaklanabilecek hata olasiligini tamamen ortadan kaldirdik.

“For” dongusunu kullanarak, bir onceki kod yonteminde oldugu gibi dizideki degerlerin sayisini degistirdi iseniz, baska herhangi bir kodu degistirmeyi hatirlamaniz gerekmez.

“For” dongulerinin guvenilir ve kisa olmasi, onlari Rust’ta en yaygin kullanilan dongu yapisi haline getirir. Bir onceki ornekte while anahtari kullanilan ornekte oldugu gibi, bazi kodlari belirli sayida calistirmak istediginiz durumlarda bile, cogu **Rustacean** bir **“for”** dongusu kullanir. Eger bunu tersine cevirmek isterseniz yine bir “**for”** dongusu veya henuz bahsetmedigimiz baska bir yontem olan **“rev”** kullanabilirsiniz.

```
fn main() {
   for number in (1..4).rev() {
      println!("{}!", number);
   }
   println!("Tamamlandi!");
}
```

Bu kod biraz daha hos gorunuyor oyle degil mi? Basardiniz! Bu oldukca buyuk bir bolumdu: degiskenler, skaler ve bilesik veri turleri, fonksiyonlar, yorumlar, if ifadeleri ve donguler hakkinda orneklerle bilgi edindiniz! Bu bolumde anlatilan kavramlarla bazi kodlar yazarak pratik yapabilir ve bilgilerinizi pekistirebilirsiniz.

Neler Yapabilirsiniz?

* Fahrenheit ve Santigrat arasindaki sicakliklari donusturun.

* Fibonacci sayi serisini uretin.

Bir sonraki bolume devam etmeye hazir oldugunuzda Rust’ta diger programlama dillerinde bulunmayan bir kavramdan bahsediyor olacagiz.

- Sahiplik (Ownership)

- BOLUM 3 -

Sahipligi Anlamak (Understanding Ownership):

Sahiplik (Ownership), Rust'in benzersiz bir ozelligidir ve Rust dilinin geri kalani icin derin etkileri vardir. Rust'in bir cop toplayici (garbage collector)'ya ihtiyac duymadan bellek guvenligi (memory safety) garanti vermesinin arkasinda yatan kavram budur. Bu nedenle sahipligin nasil calistigini anlamak oldukca onemlidir. Bu bolumde, sahiplik ve ilgili birkac ozellik hakkinda konusuyor olacagiz.

- Odunc Alma (Borrowing)
- Dilimler (Slices)
- Rust verileri bellege nasil yerlestirir? (Rust lays data out in memory safety)

Sahiplik Nedir? (What is Ownership):

Sahiplik, bir Rust programinin bellegi nasil yonettigini kontrol eden bir dizi kuraldir. Tum programlar, calisirken bilgisayarin bellegini kullanma seklini yonetmek zorundadir. Bazi diller, program calisirken surekli olarak artik kullanilmayan bellegi arayan cop toplama mekanizmalarina (garbage collector) sahiptir; diger dillerde, programci bellegi acikca tahsis etmeli ve sonrasinda serbest birakmalidir. Rust ucuncu bir yaklasimi kullanir: Bellek, derleyicinin kontrol ettigi bir dizi kurala sahip bir sahiplik sistemi araciligiyla yonetilir. Kurallardan herhangi biri ihlal edilirse program derlenmeyecektir. Sahiplik ozelliklerinin hicbiri, calisirken programin yavaslamasina neden olmaz.

Sahiplik bircok programci icin yeni bir kavram oldugu icin alismasi biraz zaman alacaktir. İyi haber su ki, Rust ve sahiplik sisteminin kurallari konusunda ne kadar deneyimli olursaniz, dogal olarak guvenli ve verimli kod gelistirmeyi o kadar kolay bulacaksiniz. Devam edin!

Sahipligi anladiginizda, Rust'i benzersiz kilan ozellikleri anlamak icin saglam bir temele sahip olacaksiniz. Bu bolumde, cok yaygin bir veri yapisina odaklanan bazi ornekler uzerinde calisarak sahipligi ogreneceksiniz: dizeler (strings).

Ancak bundan once yigin ve istif (heap and stack) kavrami uzerinde biraz duracagiz ve bazi seyleri aciklamaya calisacagiz.

Yigin ve Istif (Heap and Stack):

Bircok programlama dili, yigin ve istif (heap and stack) hakkinda cok sik dusunmenizi gerektirmez. Ancak Rust gibi bir sistem programlama dilinde, bir degerin yiginda mi yoksa istifte mi olmasi dilin nasil davrandigini ve neden belirli kararlar vermeniz gerektigini etkiler. Mulkiyet bolumleri, bu bolumun ilerleyen kisimlarinda yigin ve istifle ilgili olarak aciklanacaktir, bu nedenle burada bir hazirlik asamasinda olan kisa bir aciklama yapmamiz gerekir.

Hem yigin hem de istif (ya da obek), calisma zamaninda (runtime) kodunuzun kullanabilecegi bellek parcalaridir, ancak bunlar farkli sekillerde yapilandirilmistir. Istif (stack) degerleri aldigi sirayla saklar ve degerleri ters sirada kaldirir. Buna son giren ilk cikar (last in first out) denir. Bir tabak dusunun: daha fazla tabak eklediginizde, onlari istifin ustune koyarsiniz ve bir tabaga ihtiyaciniz oldugunda ustten bir tane alirsiniz. Ortadan veya alttan plaka eklemek veya cikarmak da ise yaramaz! Veri (data) eklemeye ise istife iteleme (poppong off the stack) denir. Istiflenen ve depolanan tum verilerin bilinen, sabit bir boyutu olmalidir. Derleme zamaninda bilinmeyen bir boyuta veya degisebilecek bir boyuta sahip veriler bunun yerine bu istiflemenin en ustunde olmalidir.

Yigin (heap) daha az organizedir: yigina veri koydugunuzda, belirli bir miktar alan talep edersiniz. Bellek ayirici (memory allocator) yiginda yeterince buyuk bos bir nokta bulur, onu kullanimda olarak isaretler ve o konumun adresi olan bir isaretci (pointer) dondurur. Bu isleme yigin uzerinde tahsis (allocationg on the heap) denir ve bazen sadece tahsis (allocating) olarak kisaltilir.

Degerleri yigina itmek tahsis olarak kabul edilmez. Yigin isaretcisi (pointer) bilinen, sabit bir boyut oldugundan, isaretciyi yiginda saklayabilirsiniz, ancak gercek verileri istediginizde isaretciyi izlemelisiniz. Bir restoranda oturdugunuzu dusunun. Girdiginizde grubunuzdaki kisi sayisini belirtirsiniz ve gorevliler herkese uyan bos bir masa bulup sizi oraya yonlendiriyor. Grubunuzdan birisi gec gelirse, siiz bulmak icin nerede oturdugunuzu sorabilir.

Yigina itme, yiginda tahsis etmekten daha hizlidir, cunku ayirici (allocator) hicbir zaman yeni verileri depolamak icin bir yer aramak zorunda kalmaz; bu konum her zaman yiginin en ustundedir. Nispeten, yigin uzerinde alan tahsis etmek daha fazla is gerektirir, cunku tahsis edenin once verileri tutacak kadar buyuk bir alan bulmasi ve ardindan bir sonraki tahsise hazirlanmak icin defter tutma (bookkeeping) yapmasi gerekir.

Bu durumda kodunuz bir islevi cagirdiginda, isleve iletilen degerler (potansiyel olarak yigin uzerindeki adreslenmis isaretciler dahil) ve islevin yerel degiskenleri yigina aktarilir. Islev son buldugunda, bu degerler yigindan atilir.

Kodun hangi bolumlerinin yigindaki hangi verileri kullandigini takip etmek, yigindaki yinelenen veri miktarini en aza indirmek ve alaninizin bitmemesi icin yigindaki kullanilmayan verileri temizlemek, sahipligin (ownership) ele aldigi sorunlardir. Sahipligi anladiktan sonra, istif ve yigin hakkinda cok sik dusunmenize gerek kalmayacak, ancak sahipligin asil amacinin yigin verilerini yonetmek oldugunu bilmek, neden boyle calistigini aciklamaya yardimci olacak.

Mulkiyet Kurallari (Ownership Rules):

Ilk olarak, mulkiyet kurallarina bir goz atalim. Bunlari gosteren ornekler uzerinde calisirken bu kurallari mutlaka aklinizda bulundurun:

* Rust'taki her degerin sahibi olarak adlandirilan bir degiskeni daha vardir.

* Bir seferde yalnizca bir sahip olabilir.

* Sahibi kapsam disina ciktiginda ise deger duser.

Degiskenlerin Kapsami (Variable Scope):

Artik temel Rust sozdizimini geride biraktigimiza gore, orneklerde **"fn main() {"** kodunun tamamini dahil etmeyecegiz, bu nedenle takip ediyorsaniz, asagidaki ornekleri bir ana islevin icine manuel olarak koydugunuzdan emin olmalisiniz. Sonuc olarak, orneklerimiz biraz daha kisa olacak ve ortak kod yerine gercek ayrintilara odaklanmamiza izin verecek.

Ilk sahiplik ornegi olarak, bazi degiskenlerin kapsamina bakacagiz. Kapsam (scope), bir ogenin gecerli oldugu bir program icindeki araliktir. Asagidaki degiskeni ornek olarak inceleyelim:

```
let d = "merhaba";
```

Bu d degiskeni, dize degerinin programimizin metnine sabit kodlanmis oldugu bir dize degismezi anlamina gelir. Degisken bildirildigi noktadan gecerli kapsamin sonuna kadar gecerlidir. Daha fazla detay verecek olursak let d = "merhaba"; oncesindeki satirda d degiskeni henuz bildirilmemisti ve kapsam dahilinde degildi, bildirildigi andan itibaren gecerlilik kazandi ve d degiskeni ile birseyler yapildiktan sonra kapsami tamamen bitti ve d artik gecerli bir kapsama sahip degildi. Baska bir deyisle, burada zaman icinde iki onemli nokta vardir:

* d degiskeni kapsamina girdiginde gecerlidir.

* kapsam disina cikana kadar gecerliligini korur.

Bu noktada, kapsamlar ve degiskenlerin ne zaman gecerli oldugu arasindaki iliski diger programlama dillerindekine benzer bir yaklasim sergiler. Simdi dize (string) turunu anlayarak bu kavramin uzerine insa edecegiz.

Dize Turu (String Type):

Sahiplik kurallarini gostermek icin “Veri Tipleri” bolumunde ele aldiklarimizdan daha karmasik bir veri tipine ihtiyacimiz var. Yigin, kapsami sona erdiginde ve kodun baska bir bolumunun ayni degeri farkli bir kapsamda kullanmasi gerekiyorsa, yeni, bagimsiz bir ornek olusturmak icin hizli ve onemsiz bir sekilde kopyalayabilir. Ancak obekte depolanan verilere bakmak ve Rust’in bu verileri ne zaman temizleyecegini nasil bildigini kesfetmek istiyoruz ve Dize (String) turu harika bir ornek olacak.

Bir dize degerinin programimiza sabit kodlanmis oldugu dize degismezlerini zaten gorduk. Dize degismezleri uygundur, ancak metin kullanmak isteyebilecegimiz her durum icin uygun degillerdir. Bunun bir nedeni, degismez olmalaridir. Bir digeri ise, kodumuzu yazarken her dize degeri bilinemez: ornegin, kullancii girdisini alip depolamak istersek ne olur? Bu durumlar icin Rust’in ikinci bir dize turu vardir, String (dize). Bu tur, yigina ayrilan verileri yonetir ve bu nedenle derleme zamaninda bizim icin bilinmeyen bir miktarda metin depolar, from islevini kullanarak bir dize degismezinden yeni bir dize olusturabilirsiniz. bir ornek vermek gerekirse:

```
let mut s = String::from(“merhaba”);
s.push_str(“, dunya!”);
println!(“{}”, s);
```

Yukaridaki ornekteki fark nedir? Neden String mutasyona ugratilabilir ancak degismezler olamaz?

Aradaki fark, bu iki turun bellekle nasil basa ciktigidir.

Bellek ve Tahsis (Memory and Allocation):

Bir dize degismezi durumunda, icerigi derleme zamaninda biliyoruz, bu nedenle metin dogrudan son yurutulebilir dosyaya sabit kodlanmistir. Bu nedenle dize degismezleri hizli ve verimlidir. Ancak bu ozellikler yalnizca dize degismezinin degismezliginden gelir. Ne yazik ki, derleme zamaninda boyutu bilinmeyen ve programi calistirirken boyutu degisebilecek her metin parcasi icin ikili dosyaya bir bellek blogu koyamiyoruz.

String turuyle, degisebilir, buyutulebilir bir metin parcasini desteklemek icin, icerigi tutmak icin yigin uzerinde derleme zamaninda bilinmeyen bir miktar bellek ayirmamiz gerekir.

Bellek, calisma zamaninda bellek ayiricidan talep edilmeliodir. String ile isimiz bittiginde bu hafizayi ayiriciya geri dondurmenin bir yoluna ihtiyacimiz var.

Rust farkli bir yol izler: sahip oldugu degisken kapsam disina ciktiginda bellek otomatik olarak dondurulur. Asagida, bir dize degismezi yerine bir dize kullanarak kapsam ornegimizin bir surumu verilmistir:

```
{
        let s = String::from(“merhaba”);
}
```

String’imizin ihtiyac duydugu bellegi ayiriciya dondurebilecegimiz dogal bir nokta vardir: s kapsam disina ciktiginda. Bir degisken kapsam disina ciktiginda Rust bizim icin ozel bir fonksiyon cagirir. Bu isleve **“drop”** denir ve String yazarinin bellegi geri dondurmek icin kodu koyabilecegi yerdir. Rust **drop** cagrilarinda yani kapanis kume parantezine vardiginda “}” otomatik olarak **drop** olacaktir.

Bu model, Rust kodunun yazilma sekli uzerinde derin bir etkiye sahiptir. Su anda basit gorunebilir ancak yiginda tahsis ettigimiz verileri birden cok degisken kullanmak istedigimizde, daha karmasik durumlarda kodun davranisi beklenmedi olabilir. Simdi bu durumlardan bazilarini inceleyelim.

Degiskenlerin ve Veri Etkilesiminin Yollari:

Birden cok degisken, Rust’ta ayni verilerle farkli sekillerde etkilesime girebilir. Bir tamsayi kullanan bir ornege bakalim:

```
let x = 5;
let y = x;
```

Muhtemelen yukarida neler olup bittigini tahmin edebiliriz: 5’i x’e bagla, sonra x’deki degerin bir kopyasini olustur ve onu y degiskenine bagla. Simdi iki degiskenimiz var, x ve y ve her ikisi de 5 sayisina esittir. Bu gercekten olan seydir, cunku tamsayilar bilinen, sabit bir boyuta sahip basit degerlerdir ve bu iki 5 degeri yigina itilir.

Simdi de ayni ornegin String versiyonunda ornegine bakalim:

```
let s1 = String::from("merhaba");
let s2 = s1;
```

Bu ornegin de onceki ornege benzedigi gorunuyor, bu yuzden calisma seklinin ayni olacagini varsayabiliriz. Yani ikinci satirda s1'deki degeri bir kopyasinin olusturuldugunu ve onun s2 degerine baglandigini dusunebiliriz. Ama bu tam olarak dogru degil!

Ne olduguna daha yakindan bakalim. Bir dize, uc bolumden olusur: dizenin icerigi, uzunlugu ve bir kapasiteyi tutan bellege yonelik bir isaretci (pointer). Tum bu veriler yiginda depolanir ve icerigi tutan obek uzerindeki bellek bulunur.

s1	Value
--	
ptr	0
len	5
capacity	5

length (len) String iceriginin su anda bayt cinsinden ne kadar bellek kullandigini gosterir. Kapasite (capacity), dizenin ayiricidan aldigi bayt cinsinden toplam bellek miktari olarak gosterir. Uzunluk ve kapasite arasindaki fark onemlidir, ancak bu baglamda degil, bu nedenle simdilik kapasiteyi goz ardi etmekte fayda var.

Yukaridaki ornekte oldugu gibi s1 s2 degerine atandiginda, String verileri kopyalanir, yani yigindaki isaretci uzunluk ve kapasite kopyalanir. Isaretcinin basvurdugu obek uzerindeki verileri kopyalamayiz. Baska bir deyisle bellekteki verilerle ilgilenmiyoruz yigin itimindeki uc bilgi ile ilgileniyoruz. Simdi bunu gercekten gorelim:

```
let s1 = String::from("merhaba");
let s2 = s1;
println!("{}, dunya!", s1);
```

$ cargo run

Compiling ownership v0.1.0 (/Users/zgr/projects/ownership)

error[E0382]: borrow of moved value: `s1`

Diger programlama dillerinde calisirken sig kopyalama (shallow copy) ve derin kopyalama (deep copy) terimlerini duydu iseniz verileri kopyalamadan isaretciyi, uzunlugu ve kapasiteyi kopyalama kavrami muhtemelen sig bir kopya olusturmaya benziyor. Ancak Rust ayni zamanda ilk degiskeni de gecersiz kildigi icin, onu sig bir kopya olarak adlandirmak yerine hareket olarak bilinir.

Bu bizim yukarida karsilastigimiz sorunumuzu cozuyor! Yalnizca s2 gecerli oldugunda, kapsam disina ciktigiginda tek basina bellegi bosaltir ve isimiz biter. Rust, verilerinizin derin kopyalarini asla otomatik olarak olusturmaz. Bu nedenle herhangi bir otomatik kopyalamanin calisma zamani performansi acisindan kolay oldugu varsayilabilir.

Degiskenlerin ve Veri Etkilesiminin Yollari Klon (clone):

Sadece yigin verilerini degil, String'in yigin verilerini de derinlemesine kopyalamak istiyorsak, **klon (clone)** adi verilen ortak bir yontem kullanabiliriz. Iste yukaridaki hata ile ilgili bir klon yaklasimi:

```
let s1 = String::from("merhaba");
let s2 = s1.clone();
println!(s1 = {}, s2 = {}", s1 s2);
```

Yukaridaki kod gayet iyi calisir ve yigin verilerinin kopyalandigini daha dogrusu klonlandigini acikca gosterir. Bir klonlama cagrisi gordugunuzde, bazi rastgele kodlarin yurutuldugunu ve bu kodun pahali olabilecegini bilirsiniz. Bu, farkli bir seyin olup bittiginin gorsel bir gostergesidir.

Yigin bazli Veri Kopyalama (copy):

Derleme zamaninda boyutu bilinen tamsayilar gibi turlerin tamamen yiginda saklanmasidir, bu nedenle gercek degerlerin kopyalari hizli bir sekilde olusturulur. Simdi bir kod ornegi gorelim:

```
let x = 5;
let y = x;
println!("x = {}, y = {}", x, y);
```

Yukaridaki ornekte y degiskenini olusturdukdan sonra x'in gecerli olmasini engellemek istememiz icin hicbir neden olmadigi anlamina gelir. Baska bir deyisle, burada derin ve sig kopyalama arasinda bir fark yoktur, bu nedenle klon cagirmak normal sig kopyalamadan farkli bir sey yapmaz ve bunu disarida birakabiliriz.

Mulkiyet ve Fonksiyonlar (Ownership and Functions):

Bir fonksiyona deger aktarmanin semantigi, bir degiskene deger atamanin semantigine benzer. Bir fonksiyona bir degisken iletmek tipki atamada oldugu gibi tasinir ve kopyalanir. Degiskenlerin nerede kapsam icine girip nerede kapsam disinda kaldigini gosteren bazi aciklamalar icin bir ornek yapalim:

```
fn main() {
    let s = String::from("merhaba");    // s degiskeni icin kapsam basladi
    takes_ownership(s);                 // s degeri fonksiyona tasiniyor
    let x = 5;                          // x degiskeni icin kapsam basladi
    makes_copy(x);                      // x degeri fonksiyona tasiniyor
}                                       // x icin kapsam bitti ancak s degeri tasindigi icin ozellesti

fn takes_ownership(some_string: String) {   // some_string kapsami basladi
    println!("{}", some_string);
}                                           // some_string kapsami bitti yani drop cagrildi.

fn makes_copy(some_integer: i32) { // some_integer kapsami basladi
    println!("{}", some_integer);
}                                       // some_integer kapsami bitti ve ozel bir durumu yok.
```

Yukaridaki ornekte take_ownership cagrisindan sonra s kullanmak isterseniz Rust bir derleme zamani (compile time) hatasi verir. Bu statik kontroller bizi hatalardan koruyacaktir. Bunlari nerede kullanabileceginizi ve mulkiyet kurallarinin bunu yapmanizi nerede engellendegini gormek icin s ve x kullanan ana kodu eklemeyi deneyin.

Donus Degerleri ve Kapsam (Return Values and Scope):

Donen degerler de mulkiyeti aktarabilirler. Asagidaki ornegi incelerseniz fonksiyonun dondurdugu degerlerin sahipligi aktardigini gorursunuz.

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership dongusu cagrildi ve s1 atandi
    let s2 = String::from("merhaba");     // s2 icin kapsam basladi
    let s3 = takes_and_gives_back(s2);  // s2 donus degeri s3'e atandi
}                                                  // s3 icin kapsam bitti ve drop cagrildi

fn gives_ownership() -> String {                 // gives_ownership tasindi
    let some_string = String::from("yours");          // some_string icin kapsam basladi
    some_string                                       // some_string donus degeri aliniyor
}

fn takes_and_gives_back(a_string: String) -> String { // a_string icin kapsam basladi
    a_string  // a_string  donduruluir ve cagiran fonksiyona tasinir
}
```

Yukarida da goruldugu uzere bir degiskenin mulkiyeti her seferinde ayni kalibi takip eder: baska bir degiskene bir deger atama onu hareket ettirir. Yigin uzerindeki verileri iceren bir degisken kapsam disina ciktiginda, verilerin sahipligi baska bir degiskene tasinmadikca deger damla damla temizlenir.

Referanslar ve Odunc Alma (References and Borrowing):

Asagidaki ornegi inceleyelim. Bu ornekte uzunluk_hesaplama icine tasima yapildigindan, uzunluk_hesaplama cagrisindan sonra yine de dizeyi kullanabiliriz. Bunun yerine, String degerine bir referans saglayabiliriz. Referans, bir isaretci gibidir, cunku o adreste saklanan ve baska bir degiskene ait olan verilere erismek icin takip edebilecegimiz bir adrestir. Isaretciden farkli olarak, referansin belirli bir turun gecerli bir degerine isaret etmesi garanti edilir. Degerin sahipligini almak yerine parametre olarak bir nesneye referansi olan bir uzunluk_hesaplama islevini nasil tanimlaaycaginiz ve kullanacaginizi nasil yapacaginizi gorelim:

```
fn main() {
    let s1 = String::from("merhaba");
    let len = uzunluk_hesaplama(&s1);
    println!("degiskenin '{}' uzunlugu {}.", s1, len);
}

fn uzunluk_hesaplama(s: &String) -> usize {
    s.len()
}
```

Oncelikle, degisken bildirimindeki tum tanimlama grubu kodunun ve islev donus degerinin kaybolduguna dikkat edin. Sonrasinda &s1 uzunluk_hesaplama iletilir ve taniminda String yerine &string degerini alinir. Bu ve isaretleri referanslari temsil eder ve sahipligini almadan bazi degerlere basvurmanizi mumkun kilar.

Referans Kurallari (The Rules of References):

Referanslar hakkinda tartistiklarimizin bir ozeti sudur:

* Herhangi bir zamanda bir degisken referansiniz veya herhangi bir sayida degismez referansiniz olabilir.

* Referanslar her zaman gecerli olmalidir.

Simdi farkli bir referans turune bakacagiz: Dilimler! (**Slices**).

Dilim Turleri (The Slice Types):

Dilimler (Slices), koleksiyonun tamami yerine bir koleksiyondaki bitisik bir öge dizisine basvurmamiza izin verir. Bir dilim aslinda bir tur referanstir ve dolayisi ile sahipligi yoktur.

Iste bir programlama problemi: bir dizge alan ve bu dizgede buldugu ilk kelimeyi donduren bir fonksiyon yazalim. Islev dizgede bir bosluk bulamazsa, tum dize bir kelime olmalidir, bu nedenle tum dize dondurulmelidir. Dilimlerin cozecegi problemi anlamak icin dilimleri kullanmadan bu fonksiyonu nasıl yazacagimiz uzerinde calisalim:

```
fn ilk_kelime(s: &String) -> ?
```

Burada ilk_keliome islevi, parametre olarak bir &String parametresine sahiptir. Mulkiyet istemiyoruz, bu yuzden bu iyi. Ama neyi iade etmeliyiz? Bir dizinin bir parcasi hakkinda konusmanin gercekten bir yolu yok. Ancak, bir boslukla gosterilen kelimenin sonunun dizinini dondurebiliriz.

```
fn ilk_kelime(s: &String) -> usize {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }
    s.len()
}
```

Yukarida String parametresine bir bayt indeks degeri donduren ilk_kelime fonksiyonunu goruyorsunuz. String ogesini elemanlari bazinda gozden gecirmeniz ve bir degerin bosluk olup olmadigini kontrol etmemiz gerektiginden as_bytes metodunu kullanarak String'imizi bir bayt dizisine donusturecegiz. Ardindan yineleme metodunu kullanarak bayt dizisi uzerinde bir yineleyici (iterator) olustururuz.

Yenileyici (iterator) bir koleksiyondaki her ogeyi donduren bir metod oldugunu ve numaralandirmanin yinelemenin sonucunu sararak bunun yerine her ogeyi bir demetin parcasi olarak dondurdugunu unutmayin. Buradan dondurulen grubun ilk ogesi dizindir ve ikinci oge ogeye bir basvurudur. Bu, indeksi kendimiz hesaplamamizdan biraz daha uygundur.

Dize Dilimleri (String Slices):

Bir dize (string) dilimi, bir dizenin bir bolumune referanstir ve asagidaki gibi gorunur:

```
let s = String::from("merhaba dunya");
let merhaba = &s[0..7];
let dunya = &s[8..13];
```

Merhaba string’inin tamamina bir referans yerine, String’in [0..7] bitinde belirtilen bir kismina referans kurdugumuzu gozlemleyin. Buradaki 0 ve 7 degerleri starting_index ve end_index degerlerini belirtir. Bu sayede deger icerisinde bir aralik kullanarak dilimler (slice) olusturmus oluruz.

Ozet:

Mulkiyet, odunc alma ve dilim kavramlari derleme zamaninda Rust programlarinda bellek guvenligini saglar. Rust dili, diger sistem programlama dilleriyle ayni sekilde bellek kullaniminiz uzerinde kontrol saglar, ancak veri sahibinin kapsam disina ciktiginda bu verileri otomatik olarak temizlemesi, fazladan kod yazmaniz ve hata ayiklamaniz (garbage collection) gerekmedigi anlamina gelir.

Mulkiyet ya da sahiplik, Rust’in diger bircok bolumunun nasil calistigini etkiler, bu yuzden bu kitabin geri kalaninda bu kavramlar hakkinda cok fazla bilgi yer almayacak. Bu konuda bol bol ornekler yaparak veya bu yuzeysel aciklamalardan ziyade derinlemesine arastirma yaparak kendinizi gelistirmelisiniz.

Simdi baska bir bolume gecelim ve veri parcalarini bir yapi **(struct)** icinde gruplandirmaya bakalim.

-BOLUM 4-

Yapilari Tanimlama ve Ornekleme (Defining and Instantiating Structs):

Yapilar (struct), her ikisinin de birden cok iliskili degeri icermesi bakimindan “Demet Tipleri” (Tuple Type) ile benzerlikler gosterir. Demetler gibi, bir yapinin parcalari farkli tiplerde olabilir. Demetlerden farkli olarak, bir yapi icinde, degerlerin ne anlama geldigini netlestirmek icin her bir veri parcasini adlandiracaksiniz. Bu adlarin eklenmesi, yapilarin demetlerden daha esnek oldugu anlamina gelir. Kisaca yapilari kullanarak bir ornegin degerlerini belirtmek veya bunlara erismek icin verilerin sirasina guvenmeniz gerekmez.

Bir yapi tanimlamak icin, **“struct”** anahtarini kullanarak tum yapiyi adlandirirsiniz. Bir yapinin adi, birlikte gruplandirilan veri parcalarinin onemini aciklamalidir. Ardindan kume parantezleri icinde alan dedigimiz veri parcalarinin adlarini ve turlerini tanimlariz. Simdi bir ornek yapalim:

```
struct Kullanici {
   aktif: bool,
   kullanici_adi: String,
   eposta: String,
   signedby: u64,
}
```

Bir yapiyi tanimladiktan sonra kullanmak icin, alanlarin her biri icin somut degerler belirterek o yapinin bir ornegini yaratiriz. Yapinin adini belirterek bir ornek olusturuyoruz ve ardindan anahtarlarin alanlarin adlari oldugu ve degerlerin bu alanlarda saklamak istedigimiz veriler oldugu anahtar: deger ciftlerini iceren kume parantezleri ekliyoruz. Alanlari “struct” icinde belirttigimiz sirayla belirtmemize gerek yok. Baska bir deyisle, yapi tanimi, tur icin genel bir sablon gibidir ve ornekler, turun degerlerini olusturmak icin bu sablonu belirli verilerle doldurur. Simdi yukaridaki yapisal ancak sablon ornegimizi kullanarak bir kullanici bildirelim:

```
fn main() {
   let kullanici1 = Kullanici {
      email: String::from("ozgurk@ieee.org"),
      kullanici_adi: String::from("ozgurk"),
      aktif: true,
     signedby: 1,
   };
}
```

Yukaridaki kod “Kullanici” yapisini sablon olarak kullanarak “kullanici1” adinda yeni bir deger elde edecektir. Buna nokta gosterimi (notation) denir. Sadece bu kullanicinin e-posta adresini istiyorsak, bu degeri kullanmak istedigimiz her yerde **“kullanici1.email”** cagirabiliriz. Ayrica nokta gosterimini (“notation” yani yapinin bir ogesine . ile erisim) kullanarak Kullanici yapisinin e-posta degerinin nasil degistigini gorebiliriz.

```
fn main() {
   let mut kullanici1 = Kullanici {
      email: String::from("ozgurk@ieee.org"),
      kullanici_adi: String::from("ozgurk"),
      aktif: true,
      signedby: 1,
   };
   kullanici1.email = String::from("ozgurk@ieee.org");
}
```

Burada yapidan turetilmis tum ornegin degistirilebilir olmasi gerektigini unutmayin; Rust, yalnizca belirli alanlari degistirilebilir olarak isaretlemenize izin vermiyor. Herhangi bir ifadede oldugu gibi, bu yeni orneginizi ortuk olarak dondurmek icin islev govdesindeki son ifade olarak yapinin yeni bir ornegini olusturabilirsiniz.

Yukaridaki ornekte verilen e-posta ve kullanici adiyla bir Kullanici ornegi donduren build_kullanici islevi cagrilir. Etkin alan true degerini signedby ise 1 degerini alir. Simdi build_kullanici islevine bakalim.

```
fn build_kullanici(email: String, kullanici_adi: String) -> Kullanici {
   Kullanici {
      email: email,
      kullanici_adi: kullanici_adi,
      aktif: true,
      signedby: 1,
   }
}
```

Yukaridaki build_kullanici islevinde oldugu gibi islevin parametrelerini yapi alanlari ile ayni adla adlandirmak mantiklidir, ancak e-posta ve kullanici adi alan adlarini ve degiskenleri tekrarlamak sikici olabilir. Yapinin daha fazla alani olsaydi, her adi tekrarlamak daha da can sikici olurdu. Neyse ki uygun bir steno (kisa gosterim) var!

Alanlarda Steno Kullanimi (Using the Field in Shorthand):

Parametre adlari ve yapi alani adlari tamamen ayni oldugundan build_kullanici'yi yeniden yazmak icin init steno soz dizimini kullanabiliriz, boylece tam olarak ayni davranir ancak e-posta ve kullanici adini tekrar etmez.

```
fn build_kullanici(email: String, kullanici_adi: String) -> Kullanici {
   Kullanici {
      email,
      kullanici_adi,
      aktif: true,
      signedby: 1,
   }
}
```

Burada, e-posta adli bir alana sahip olan Kullanici yapisinin yeni bir ornegi olusturulur. E-posta alaninin degerini build_kullanici islevi email parametresindeki degere ayarlamak istediginde email alani ve email parametresi ayni ada sahip oldugundan, email: email yerine sadece email yazmamiz yeterli olacaktir.

Yapilarin Guncellenmesi Sozdizimi ile diger orneklerden de ornekler olusturma:

(Creating instances from other instances with struct update syntax)

Baska bir ornekteki degerlerin cogunu iceren ancak bazilarini degistiren bir yapinin yeni bir ornegini olusturmak genellikle yararlidir. Bunu yapilarin guncellenmesi sozdizimini kullanarak yapabilirsiniz. Guncelleme sozdizimi (update syntax) olmadan duzenli olarak yeni bir Kullanici orneginin nasil olusturulacagini gosteriyoruz. E-posta icin yeni bir deger belirledik ancak bunun disinda kullanici1'den olusturdugumuz ayni degerleri kullanacagiz.

Hadi yapalim!

```
fn main() {
    let kullanici2 = Kullanici {
        aktif: kullanici1.aktif,
        kullanici_adi: kullanici1.kullanici_adi,
        email: String::from("ozgurk@ieee.org"),
        signedby: kullanici1.signedby,
    };
}
```

Yukarida yapi guncelleme sozdizimini kullanarak, ayni etkiyi daha az kodla elde ettik. Sozdizimi, acikca ayarlanmayan kalan alanlarin, verilen ornekteki alanlarla ayni degere sahip olmasi gerektigini belirtir.

Metod Sozdizimi (Method Syntax):

Methodlar tipki islevlere (fonksiyon) benzerler. Onlari **“fn”** anahtar sozcugu ve bir adla bildirebilirsiniz. Parametreleri ve donus degerleri olabilir ve metod baska bir yerden cagrildiginda calistirilan bazi kodlari icerirler. Islevlerden farkli olarak, metodlar bir yapi baglaminda tanimlanir ve bunlarin ilk parametresi her zaman yapinin ornegini temsil eden **“self”** olarak cagrilir.

Metod Tanimlama (Defining Methods):

Bir metod tanimlamak icin ornek olarak parametre alan ve dikdortgen (rectangle) ornegi olan alan fonksiyonunu degistirelim ve bunun yerine dikdortgen yapisinda tanimlanmis bir alan yonetimi yapalim.

$ cat src/main.rs

```
struct dikdortgen {
    width: u32,
    height: u32,
}

impl dikdortgen {
    fn alan(&self) -> u32 {
        self.genislik * self.yukseklik
    }
}

fn main() {
    let sekil1 = dikdortgen {
        genislik: 30,
        yukseklik: 50,
    };

    println!(
        "Bu dikdortgenin alani {} kare pikseldir.",
        sekil1.alan()
    );
}
```

Yukaridaki ornekte fonksiyonu dikdortgen baglaminda tanimlamak icin dikdortgen parametresinde **“impl”** (implementasyon, uygulama) blogu baslattigimiza dikkat edin. Bu **“impl”** blogundaki her sey dikdortgen turuyle iliskilendirilecektir. Ardindan, alan islevi impl kume parantezleri icinde haraket eder ve ilk parametreyi imzada ve govdenin her yerinde **“self”** olacak sekilde degistiririz. Ana alanda, alan islevini cagirdigimiz ve arguman olarak sekil1’i ilettigimiz yerde, bunun yerine dikdortgen ornegimizde alan yonetimini cagirmak icin yontem sozdizimini **(method syntax)** kullaniriz.

Ozet:

Yapilar, etki alaniniz icin anlamli olan ozel turler olusturmaniza olanak tanir. Yapilari kullanarak, ilioskili veri parcalarini birbirine bagli tutabilir ve kodunuzu netlestirmek icin her parcayi adlandirabilirsiniz. **“impl”** bloklarinda, turunuze iliskili islevleri tanimlayabilirsiniz ve yontemler, yapilarinizin bir orneklerinin sahip oldugu davranisi belirlemenize izin veren bir tur iliskili islevdir.

Ancak ozel turler olusturmanin tek yolu yapilar degildir. Bu metnin ikinci versiyonunda **“Enum” (Enumeration)** optimizasyon konusunu isliyor olacagiz.

Not: Enum (Enumeration) neden su anda yok? Cunku yazar hem konuyu daha derinlemesine anlamak hem de dogru Turkce karsiliklari konusunda calismalarina devam ediyor.

Hata Yonetimi (Error Handling):

Rust’in guvenilirlige olan bagliligi, hata islemeye kadar uzanir. Hatalar, yazilimda hayatin bir gercegidir, bu ndenle Rust, bir seylerin yanlis gittigi durumlari ele almak icin bir dizi ozellige sahiptir. Cogu durumda, Rust, bir hata olasiligini kabul etmenizi ve kodunuz derlenmeden once bazi islemler yapmanizi gerektirir. Bu gereksinim, kodunuzu uretime dagitmadan once hatalari kesfetmenizi ve bunlari uygun sekilde islemenizi saglayarak programinizi daha saglam hale getirir!

Rust, hatalari iki ana kategoriye ayirir: kurtarilabilir ve kurtarilamaz hatalar (recoverable and unrecoverable errors). Dosya yerinde bulunamadi hatasi gibi kurtarilabilir bir hata icin, sorunu kullaniciya bildirmek ve islemi yeniden denemek mantiklidir. Kurtarilamaz hatalar, bir dizinin sonunun otesindeki bir konuma erismeye calismak gibi her zaman hata belirtileridir.

Cogu baska programlama dili bu iki tur hatayi ayirt etmez ve istisnalar gibi mekanizmalar kullanarak her ikisini de ayni sekilde ele alir. Rust’in istisnalari yoktur. Bunun yerine, kurtarilabilir hatalar ve panik (panic) icin **“result<T, E>”** tipine sahiptir! Bu, programiniz kurtarilamaz bir hatayla karsilastiginda yurutmeyi durduran makrodur.

Bu noktada, bir hatadan kurtulmaya ve yurutmeyi durdurmaya karar verirken goz onunde bulundurulmasi gereken noktalari isleyecegiz.

Panik ve Kurtarilamaz Hatalar (Unrecoverable Errors with panic!)

Bazen kodunuzda kotu seyler olur ve bu konuda yapabileceginiz hicbir sey yoktur. Bu durumlarda Rust panige kapilir! Bunun icin bir **“panic!”** makrosuna sahiptir. Eger bu makro yurutulurse, programiniz bir hata mesaji yazdiracak, cozecek ve yigini temizleyecek son olarak programdan cikacaktir. Bu genellikle bir tur hata tespit edildiginde ve programcinin hatayi nasil ele alacaginin net olmadigi durumlarda ortaya cikar.

Varsayilan olarak, bir panik meydana geldiginde program cozulmeye baslar, bu da Rust’in yigini geri aldigi ve karsilastigi her islevden gelen verileri temizledigi anlamina gelir. Ancak bu geri donus ve temizlik cok istir. Alternatif, programi temizlemeden sonlandirarak hemen iptal etmektir. Ancak bu durumda programin kullandigi bellegin isletim sistemi tarafindan temizlenmesi gerekecektir.

Simdi basit bir programda **“panic!”** makrosunu kullanmayi deneyelim.

$ cat src/main.rs

```
fn main () {
        panic!(“crash and burn”);
}
```

Eger bu programi yurutturseniza asagidaki gibi bir hata alacaksiniz:

$ cargo run

thread 'main' panicked at 'crash and burn', src/main.rs:2:5

note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

Burada gordugunuz sey bir panic! Cagrisidir. Son iki satirda yer alan hata mesajina neden olur. Ilk satir panik mesajimizi ve kaynak kodumuzda panigin meydana geldigi yeri gosterir. Ikinci satir ise besinci karakteri isaret eder. Kisaca bize raporlanan satir kodumuzun bir parcasidir ve o satira giderseniz panic! makrosunu gorursunuz.

~~SON~~