T.C.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

PAZARYERi MESLEK YÜKSEKOKULU

BİLGİSAYAR PROGRAMCILIĞI BÖLÜMÜ

JULİA NESNE YÖNELİMLİ PROGRAMLAMA DİLİ

Öğrenci Zeliha Alcık

Öğrenci Numarası

ÖĞRETİM GÖREVLİSİ

ZAFER SERİN

BLP108 Masaüstü Uygulama Geliştirme Ödevi

BİLECİK 2026

İÇİNDEKİLER

1 -Julia'ya Giriş

Günümüzde bilimsel hesaplama, veri bilimi ve yapay zeka gibi alanlarda artan performans ve esneklik ihtiyaçlarına yanıt olarak geliştirilen, julia programlama dili, kullanıcılara 'C/C++ hızı sunarken Python kadar kolay kullanımlı kalmak üzere tasarlanmış, yüksek düzeyli, tam zamanında derlenmiş, dinamik bir programlama dili' olduğunu vurgulayarak öne çıkmaktadır Bu bölüm,kapsamlı bir giriş niteliğinde olup,bu dilin temel özelliklerinden kullanım alanlarına, temel programlama kavramlarından kod çalıştırma yöntemlerine kadar geniş bir perspektif sunmayı amaçlamaktadır. Ayrıca, farklı işletim sistemlerinde Julia'nın nasıl kurulacağı ve popüler geliştirme ortamı Visual Studio Code'un Julia ile nasıl entegre edileceği gibi pratik bilgiler de bu bölüm içerisinde yer almaktadır. Bu giriş, Julia ile etkili bir şekilde çalışmaya başlamak için sağlam bir temel oluşturmayı hedeflemektedir

1.1 Julia'nın Temel Özellikleri

Bu bölümde, bilimsel hesaplama ve veri bilimi alanlarında hızla yükselen Julia programlama dilinin ayırt edici ve onu güçlü kılan temel nitelikleri tanıtılacaktır. Julia'nın ne olduğu ve benzersiz mimarisinin yanı sıra, hangi alanlarda etkin bir şekilde kullanılabileceği de bu başlık altında detaylıca ele alınarak, dilin genel yapısına dair kapsamlı bir bakış açısı sunulacaktır.

1.1.1 Julia Nedir?

Julia yüksek başarımlı üst düzey bir programlama dilidir. Julia programlama dili MIT Universitesinden dört kişi tarafından geliştirildi ve ilk defa 2012 yılında kullanıma sunuldu. Nitelikli bir derleyici,dağıtık koşut yürütüm olanağı, sayısal hesaplamalarda yüksek doğruluk oranı ve geniş bir matematiksel işlev kütüphanesine sahip olan Julia'nın sözdizimi diğer yazılım geliştirme ortamlarında kullanılan dillerle benzerlik göstermektedir. Büyük bölümü Julia'yla yazılmış olan bu kütüphane C ve Fortran kullanılarak geliştirilmiş doğrusal cebir, rastgele sayı üretimi ve hızlı Fourier dönüşümü bileşenlerini de içermektedir. Paralel hesaplama ve Dağıtık hesaplama yöntemlerini içermektedir. Genel olarak Veri bilimi, Yapay zeka ve bilimsel hesaplama alanlarında sıkça tercih edilmektedir.Ayrıca julia programlama dilinin dosya uzantısı ".jl" dir.

1.2 Julia'nın Kullanım Alanları

Sayısal Analiz ve Bilimsel Hesaplama: Julia, yüksek hassasiyetli aritmetik, doğrusal cebir, diferansiyel denklemlerin çözümü ve optimizasyon gibi sayısal görevler için optimize edilmiştir.Akademisyenler, karmaşık matematiksel modelleri verimli bir şekilde uygulamak ve analiz etmek için Julia'yı kullanır.Örnek uygulamalar arasında mühendislik simülasyonları, fiziksel sistem modellemeleri ve finansal modellemeler bulunur.

Veri Bilimi ve Makine Öğrenimi: Julia'nın hızı ve büyük veri kümelerini işleme yeteneği, onu veri bilimi ve makine öğrenimi araştırmaları için uygun kılar.Flux.jl, Knet.jl gibi kütüphaneler derin öğrenme modellerinin geliştirilmesine olanak tanır.Veri manipülasyonu, görselleştirme ve istatistiksel analiz için çeşitli paketler (örn. DataFrames.jl, Plots.jl, Distributions.jl) mevcuttur.

Optimizasyon: Julia, doğrusal, doğrusal olmayan ve karma tamsayılı optimizasyon problemlerini çözmek için güçlü araçlar sunar (örn. JuMP.jl).Operasyonel araştırma, ekonomi, mühendislik ve yöneylem araştırması gibi alanlarda optimizasyon algoritmalarının geliştirilmesi ve test edilmesi için kullanılır.

Simülasyon ve Modelleme: Karmaşık sistemlerin dinamiklerini modellemek ve simüle etmek için Julia kullanılır.Biyoloji,kimya,ekoloji ve sosyal bilimler gibi çeşitli disiplinlerde simülasyon çalışmaları yürütülür.DifferentialEquations.jl gibi paketler, diferansiyel denkleme dayalı modellerin verimli bir şekilde çözülmesini sağlar.

Yüksek Performanslı Hesaplama (HPC): Julia, paralel ve dağıtık hesaplama yetenekleri sayesinde HPC kümelerinde kullanılabilir.Büyük ölçekli bilimsel problemlerin ve simülasyonların daha hızlı çözülmesine olanak tanır.MPI.jl ve Distributed.jl gibi paketler, çoklu işlemcilerde veya düğümlerde paralel programlama yapmayı kolaylaştırır.

Ekonomi ve Finans: Ekonomik modellerin simülasyonu, ekonometrik analizler ve finansal risk modellemeleri için Julia tercih edilmektedir.Hızlı hesaplama yeteneği, büyük finansal veri setlerinin işlenmesinde ve karmaşık ekonomik modellerin çözülmesinde önemlidir.

1.2.1 Programlama Dili Seviyeleri

Programlama dilleri, makine koduna yakınlıklarına göre başlıca üç seviyeye ayrılmaktadır:

Yüksek Seviyeli Diller:İnsan diline en yakın ve soyutlama düzeyi en yüksek olan dilleri ifade eder. Bu diller, okunaklılık ve yazım kolaylığı sunar. Örnek olarak Julia, Python, Java, JavaScript, Ruby ve PHP gösterilebilir.

Orta Seviyeli Diller: Anlaşılması ve yazılması açısından ne yüksek seviyeli ne de düşük seviyeli diller kadar karmaşık olan dillerdir. Hem insan diline göre daha anlaşılır hem de donanım seviyesine daha yakın kontrol imkanı sunarlar. Örnek olarak C ve C++ verilebilir.

Düşük Seviyeli Diller: Makine diline en yakın olan programlama dilleridir. Bu diller, donanım üzerinde doğrudan ve hassas kontrol imkanı sağlar ancak yazılması ve anlaşılması daha zordur. Örnek olarak Assembly dili gösterilebilir.

Bir programlama dili insan diline ne kadar yakınsa, o kadar anlaşılır ve yazılması kolay olurken; makine diline ne kadar yakınsa, bilgisayar için o kadar hızlı çalışır ve işlenmesi kolay olur. Bilgisayarlar kendi makine dillerine yakın olan dillerde daha yüksek performans sergilerler.

1.3 Temel Kavramlar (Basics)

1.4 Derleyici ve Yorumlayıcı:Dönüştürücülerin Temel İşlevleri ve Farkları

Derleyiciler ve yorumlayıcılar, programlama dillerinde yazılmış kaynak kodunu, bilgisayarların doğrudan işleyebileceği makine koduna dönüştürme işlevi gören temel yazılım araçlarıdır. Her ikisi de bu dönüşüm sürecini üstlenerek, programlama dillerinin insan tarafından yazılabilirliğini ve bilgisayar tarafından yürütülebilirliğini sağlayan birer çevirmen görevi üstlenirler. Bu dönüştürme işlemi,programlama dilinde ifade edilen mantığın, bilgisayarın merkezi işlem birimi (CPU) tarafından kontrol edilmesini sağlayan ikili (0 ve 1) makine dili talimatlarına dönüştürülmesini içerir. Makine kodu,doğrudan CPU'yu yöneten temel komut setlerinden oluşan düşük seviyeli bir bilgisayar dilidir.

1.4.1 Derleyiciler

Derleyici, bütün bir programın kaynak kodunu tek seferde analiz eden ve programı çalıştırmadan önce tamamını makine koduna dönüştüren bir yazılımdır. Derlenmiş kod, bellekte daha fazla yer kaplayan bir çalıştırılabilir dosya (.exe, .app vb.) olarak üretilir.

Derleyicinin Avantajları:

Yüksek Performans: Derlenmiş kod, yorumlanmış koda kıyasla genellikle daha hızlı çalışır, çünkü tüm kod dönüştürme süreci programın yürütülmesinden önce tamamlanır ve çalışma anında ek bir çeviriye ihtiyaç duyulmaz.

Gelişmiş Güvenlik: Derleyiciler, uygulamaların güvenliğini artırmaya yardımcı olabilir. Derlenmiş kod,kaynak kodunu doğrudan içermediği için, tersine mühendislik yoluyla programın orijinal mantığına erişmek daha zordur.

Hata Ayıklama Araçları: Derleyiciler genellikle entegre Hata Ayıklama (Debugging) araçları sunarak, geliştirme sürecindeki sözdizimi ve bazı anlamsal hataların tespit edilmesini ve düzeltilmesini kolaylaştırır.

Derleyicinin Dezavantajları:

Sınırlı Hata Yakalama: Derleyici, yalnızca sözdizimi hatalarını ve derleme zamanında tespit edilebilen belirli anlamsal hataları yakalayabilir. Çalışma zamanı hatalarını tespit edemez.

Uzun Derleme Süresi: Özellikle büyük ölçekli projelerde, tüm kod tabanının derlenmesi uzun zaman alabilir, bu da geliştirme döngüsünü yavaşlatabilir.

1.4.2 Yorumlayıcılar

Yorumlayıcı, programın kaynak kodunu satır satır veya komut komut okuyarak, her komutu anında makine koduna çevirir ve çalıştırır. Bu süreç, programın çalıştırılmasıyla eş zamanlı olarak gerçekleşir. Yorumlayıcılar genellikle bellekte daha az yer kaplar çünkü bütün bir çalıştırılabilir dosya üretmezler.

Yorumlayıcının Avantajları:

Kolay Hata Ayıklama: Yorumlanan dillerde hata ayıklama süreçleri genellikle daha kolaydır, çünkü yorumlayıcı hatanın bulunduğu satırı anında belirtebilir ve geliştiricinin sorunlu kısmı tek tek incelemesine olanak tanır.

Platform Bağımsızlığı: Yorumlanan diller genellikle farklı işletim sistemlerinde veya donanımlarda, ilgili yorumlayıcının mevcut olması koşuluyla, doğrudan çalışabilirler.

Yorumlayıcının Dezavantajları:

Daha Yavaş Çalışma: Yorumlanan kod, derlenmiş koda kıyasla genellikle daha yavaş çalışır. Bunun nedeni, her bir komutun çalışma anında yorumlanması ve dönüştürülmesi için ek zaman gerekmesidir.

Yorumlayıcı Bağımlılığı: Yorumlanan bir programı çalıştırmak için ilgili yorumlayıcının sistemde yüklü olması gerekir. Bu da programın taşınabilirliğini ve bağımsızlığını bir ölçüde kısıtlayabilir.

1.5 Julia Kodunun Çalıştırılma Yöntemleri

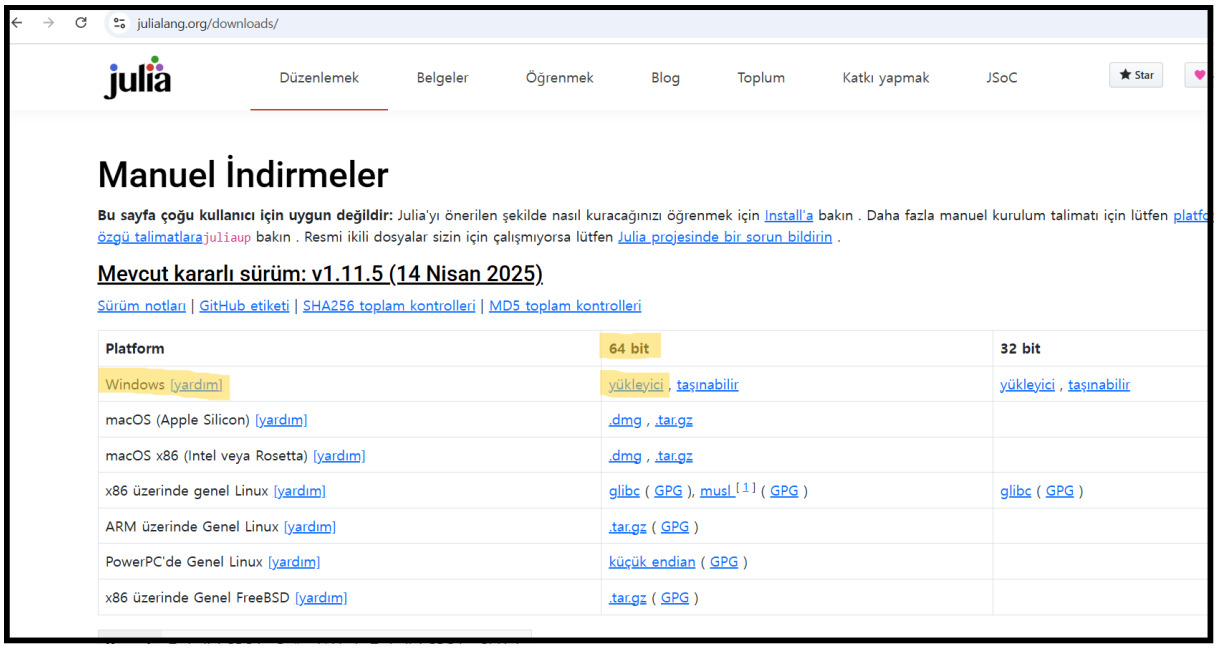
Julia, nispeten yakın zamanda geliştirilmiş, özgür ve açık kaynak kodlu bir programlama dilidir. Sözdizimi, MATLAB® ve Python gibi yaygın olarak kullanılan programlama dillerine benzerlik göstermekte olup, yüksek performans hedefiyle C benzeri hızlara ulaşabildiği belirtilmektedir. Julia geliştirmeleri için yaygın olarak tercih edilen Entegre Geliştirme Ortamlarından (IDE) biri Visual Studio Code olarak belirtilmiştir. Bu kılavuzda, Julia programlama dilinin ve Visual Studio Code (VS Code) Entegre Geliştirme Ortamının kurulum adımları ile VS Code'un Julia ile uyumlu çalışacak şekilde yapılandırılması süreçleri ele alınacaktır. Julia ve VS Code kurulum yönergeleri işletim sistemine göre farklılık göstermektedir; okuyucuların ilgili yönergelere ulaşmak için içerik tablosuna başvurması önerilir. VS Code'un Julia ile entegrasyonu için gerekli yapılandırma adımlarının ise tüm işletim sistemleri için standart olduğu belirtilmelidir.

1.6 Windows

1.6.1 Julia'yı Yükleme

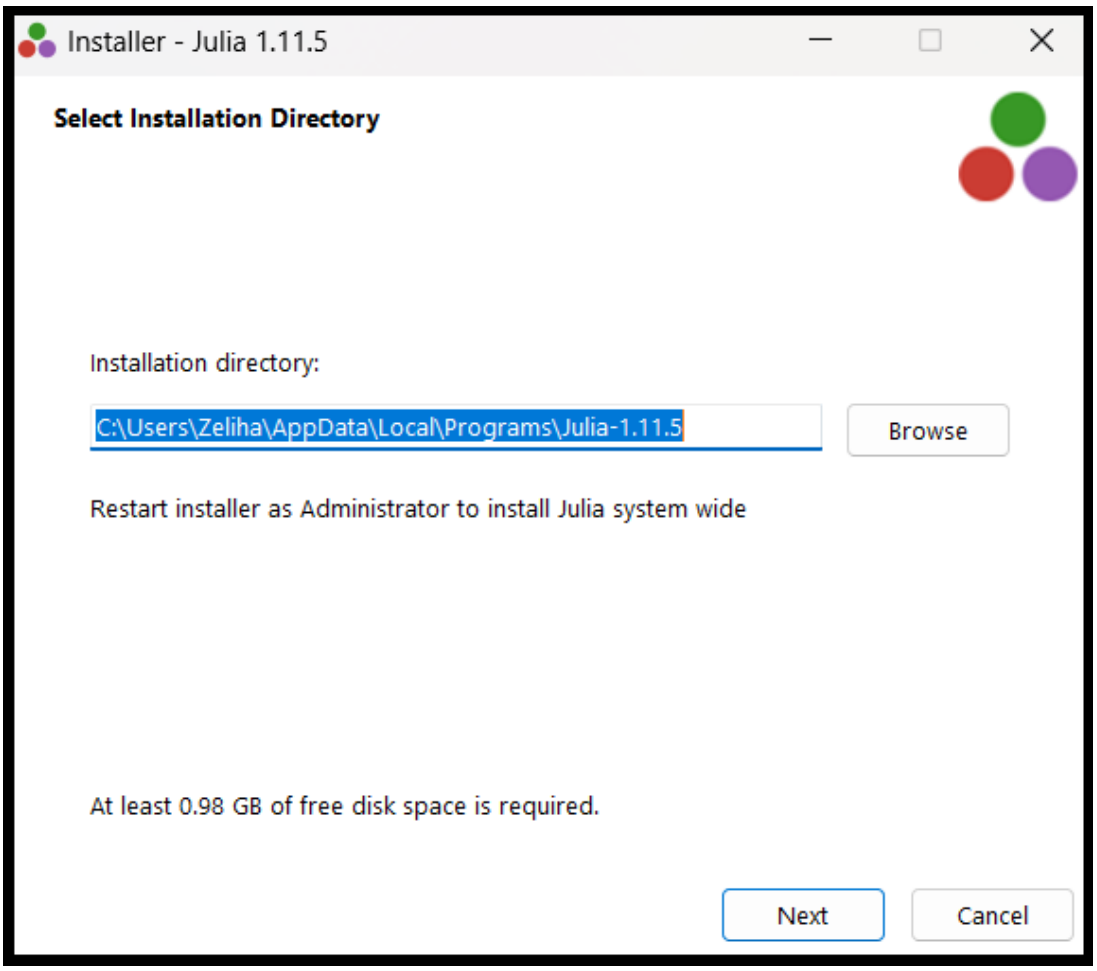
"Windows işletim sistemi üzerinde Julia programlama dilinin kurulumu, belirli adımların takip edilmesini gerektirmektedir. Bu bölümde, Julia'nın Windows ortamına kurulum sürecine yönelik detaylı yönergeler sunulacaktır. Kurulum işlemine başlamadan önce, Julia'nın resmi web sitesinden en güncel ve kararlı sürümüne ait yükleyici dosyasının temin edilmesi önemlidir. Elde edilen yükleyici dosyasının çalıştırılması ve kurulum sihirbazının yönlendirmelerine uyulmasıyla kurulum süreci tamamlanacaktır."

1. Julia'nın indirme sayfasına (https://julialang.org/downloads) erişim sağlanmalıdır.
2. 'Mevcut kararlı sürüm' başlığı altında yer alan tabloda, 'Windows' etiketi ile belirtilen satır seçilmeli ve '64-bit (yükleyici)' bağlantısı etkinleştirilmelidir. Sistem mimarisinin 32-bit olması durumunda ise ilgili 32-bit bağlantısı kullanılmalıdır.



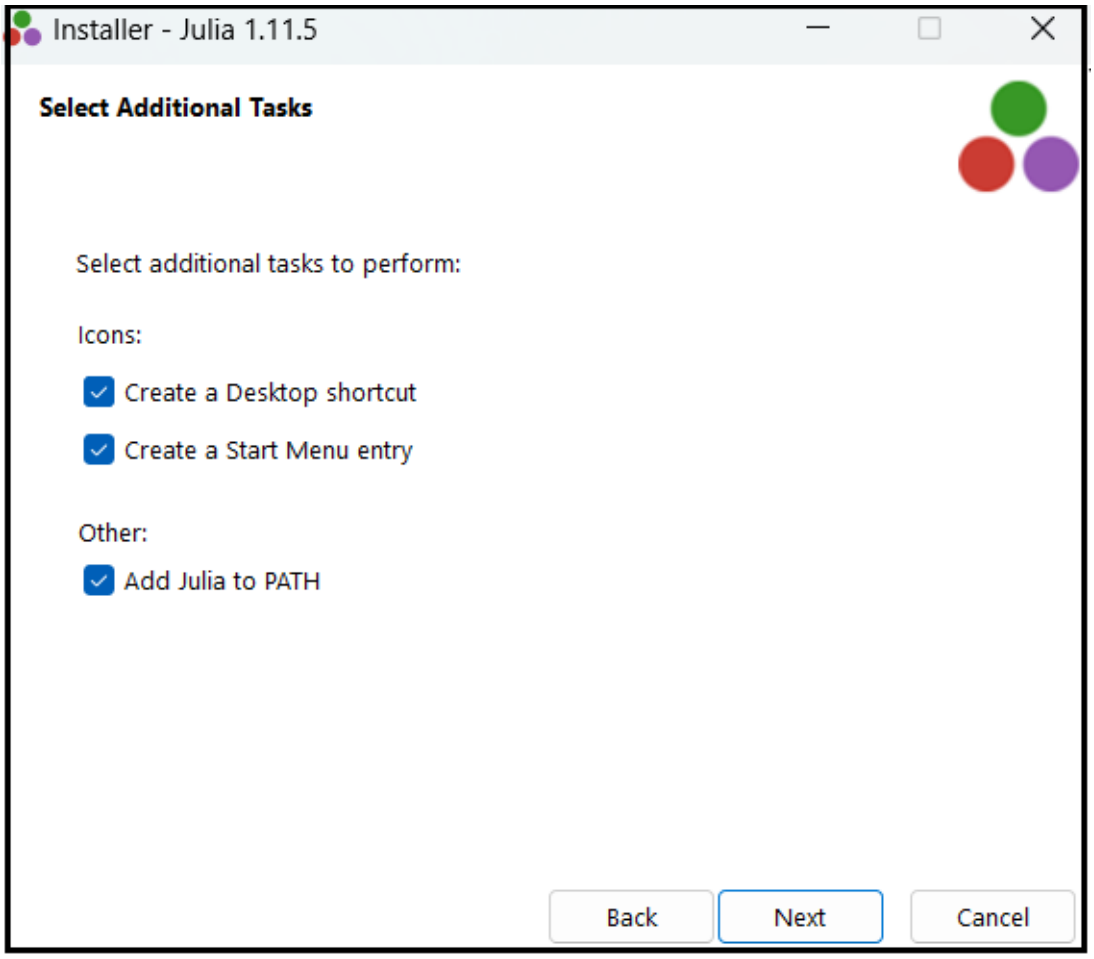
3.Kurulum programı çalıştırılmalıdır.

4.Julia'nın yükleneceği konum belirlenmelidir. (Özel bir kurulum dizini ihtiyacı bulunmadığı sürece varsayılan konum yeterli olacaktır.) Akabinde 'Next' butonuna tıklanmalıdır.



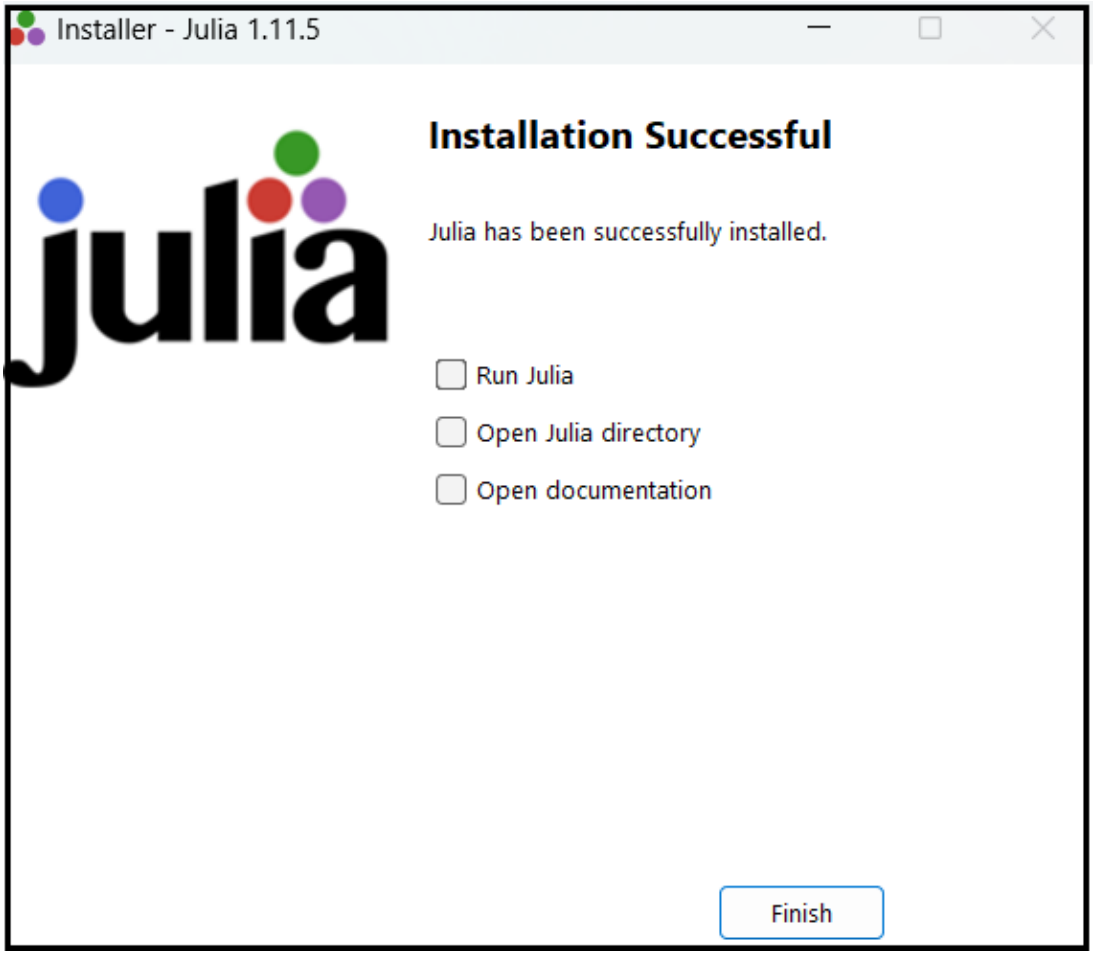
5.Ek kurulum seçenekleri belirlenmelidir.

- "Create a Desktop shortcut": Bu seçenek, Julia'nın masaüstündeki kısayolu aracılığıyla doğrudan başlatılmasını sağlar.
- "Create a Start Menu entry": Julia'nın Başlat Menüsü'ne sabitlenmesine olanak tanır. Bu sayede, Başlat Menüsü üzerinden Julia'ya hızlı erişim imkanı sunulur.
- "Add Julia to PATH": Bu özellik, Julia yürütülebilir dosyasının tam yolunun belirtilmesine gerek kalmaksızın komut satırından doğrudan çalıştırılmasına olanak tanır. Ayrıca, Visual Studio Code (VS Code) gibi entegre geliştirme ortamlarıyla daha sorunsuz bir entegrasyon sağlar. (Bu seçeneğin etkinleştirilmesi tavsiye edilmektedir.)



"Akabinde 'Next' butonuna tıklanarak kuruluma devam edilmelidir.

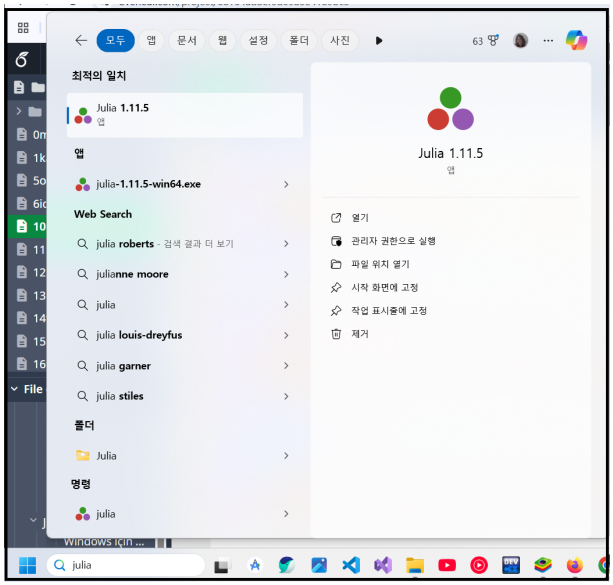
6.Kurulum işlemi başarıyla tamamlanmıştır. Kurulum penceresinden çıkmak için ’Finish’ butonuna tıklanması gerekmektedir.



1.6.2 Julia'yı Çalıştırmak

Julia programlama dilinin Windows işletim sistemi üzerinde çeşitli çalıştırma yöntemleri bulunmaktadır.

- Masaüstü kısayolu oluşturulmuş olması durumunda, Julia'yı başlatmak için ilgili kısayola çift tıklanarak işlem gerçekleştirilebilir.
- Alternatif olarak, bir Başlat Menüsü girişi mevcutsa, Julia Başlat Menüsü aracılığıyla yürütülebilir. Bu durumda, öncelikle 'Julia' ifadesinin aranması gerekmektedir.



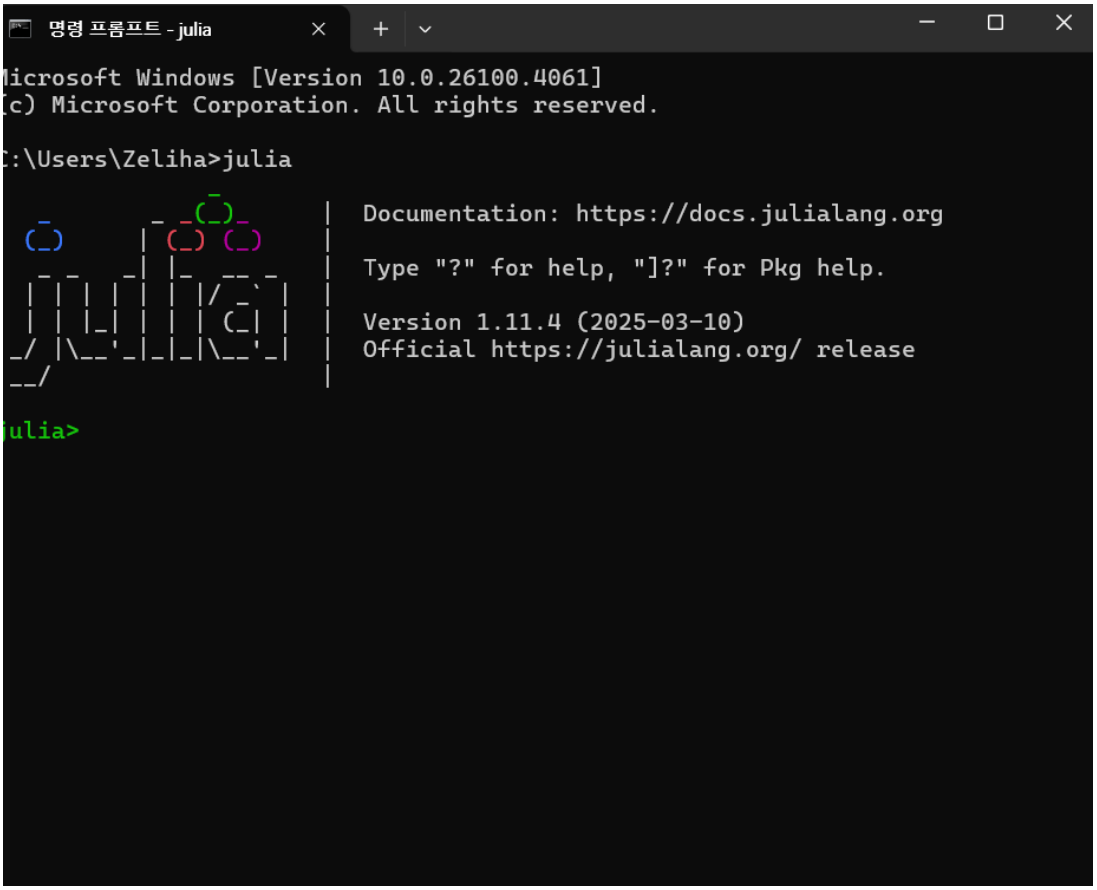
- Julia programlama dili, komut satırı arayüzleri (Komut İstemi veya PowerShell) üzerinden de yürütülebilir. Julia'nın sistemin PATH değişkenine eklenmesi durumunda, doğrudan çağrılması mümkündür.

```
C:\Users\user> julia
```

- Aksi halde, Julia yürütülebilir dosyasının tam (mutlak) veya bağıl yolu belirtilmelidir.

```
C:\Users\user> .\AppData\Local\Programs\Julia-1.11.5\bin\julia.exe
```

- Julia başarıyla başlatıldığında, yeni bir komut istemi görüntülenecektir.

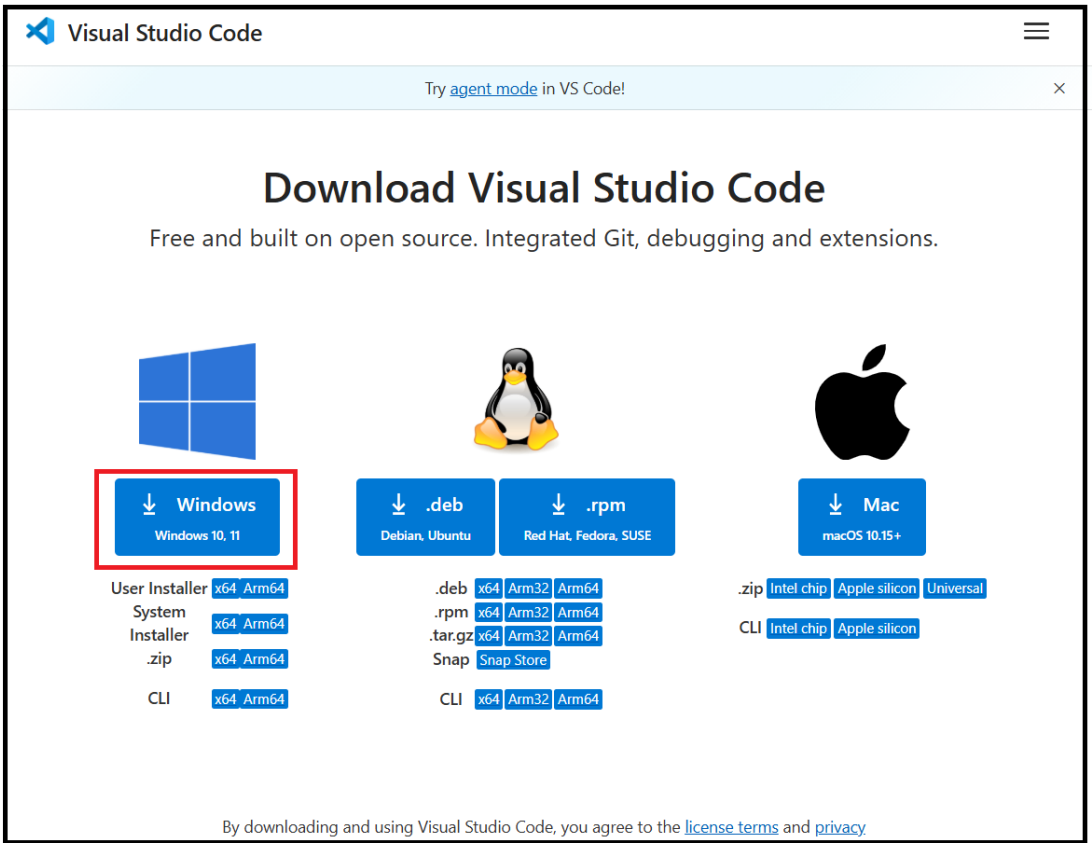


Bu bölümle birlikte, Julia'nın Windows ortamına kurulumu ve ilk çalıştırma adımları tamamlanmıştır.

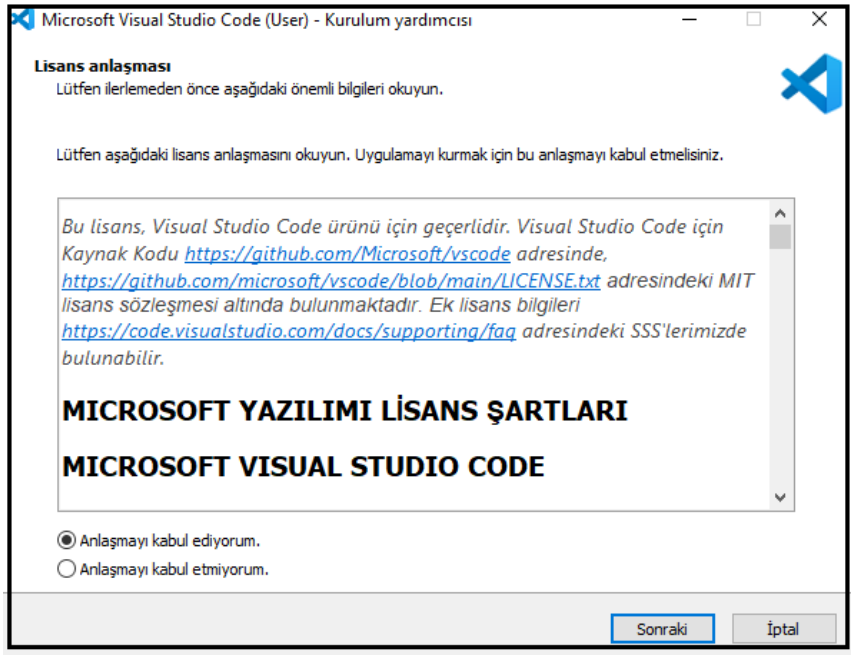
1.6.3 VS Code'u yükleme

macOS işletim sistemi üzerinde Julia programlama dilinin kurulumu, sistem mimarisine (Apple Silicon veya Intel) bağlı olarak farklılık gösterebilmektedir. Bu bölümde, Julia'nın macOS ortamına kurulum sürecine ilişkin adımlar detaylandırılacaktır. Kurulum işlemi için öncelikle Julia'nın resmi indirme sayfasından ilgili sürüme ait yükleyici dosyasının temin edilmesi gerekmektedir. Ardından, indirilen dosyanın çalıştırılması ve kurulum sihirbazındaki yönergelerin takip edilmesiyle süreç tamamlanacaktır.

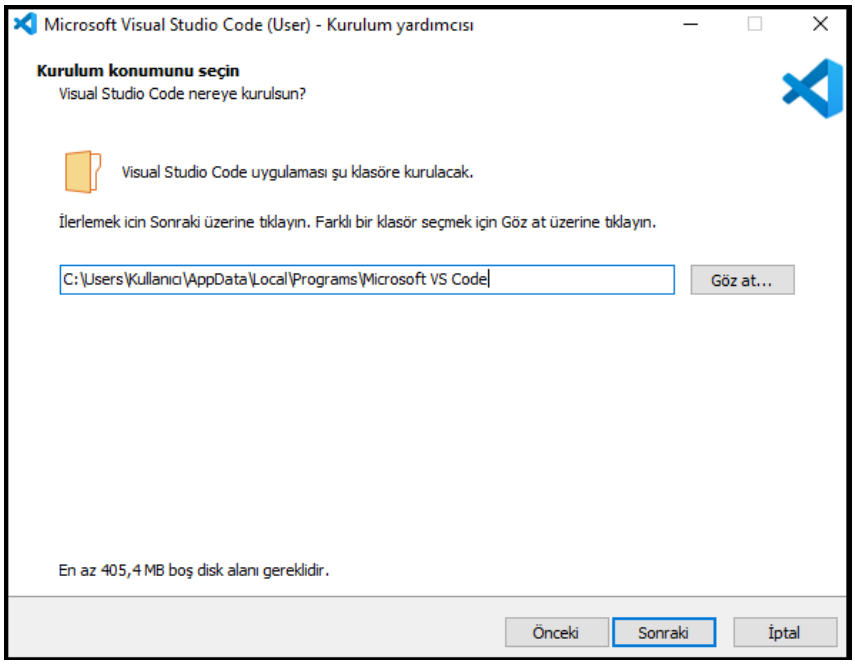
1. Visual Studio Code'un indirme sayfasına (https://code.visualstudio.com) erişim sağlanmalıdır.
2. 'Windows için İndir' butonuna tıklanmalıdır.



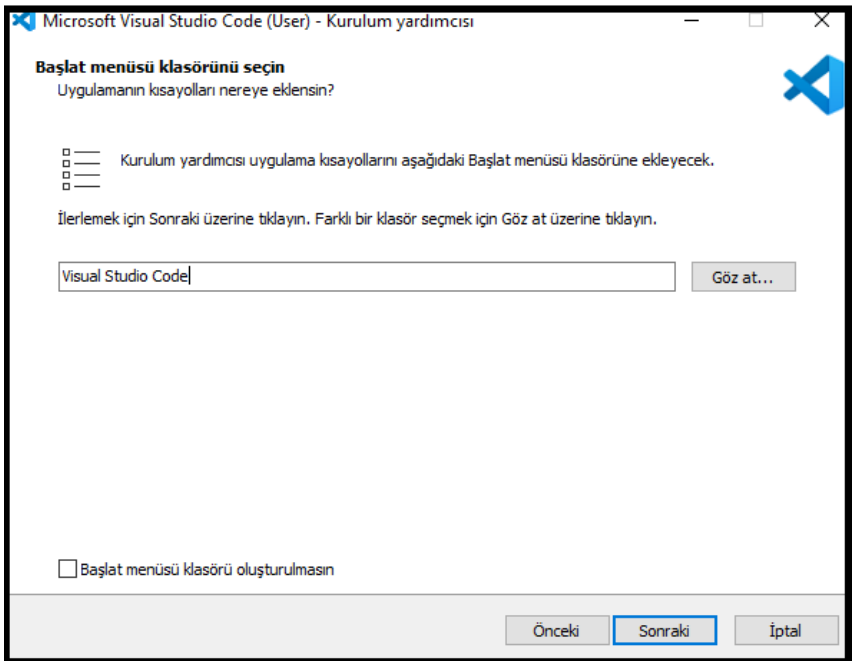
3. Kurulum programı çalıştırılmalıdır.
4. ’Sözleşmeyi kabul ediyorum’ seçeneği etkinleştirilmeli ve ardından ’İleri’ butonuna tıklanmalıdır.



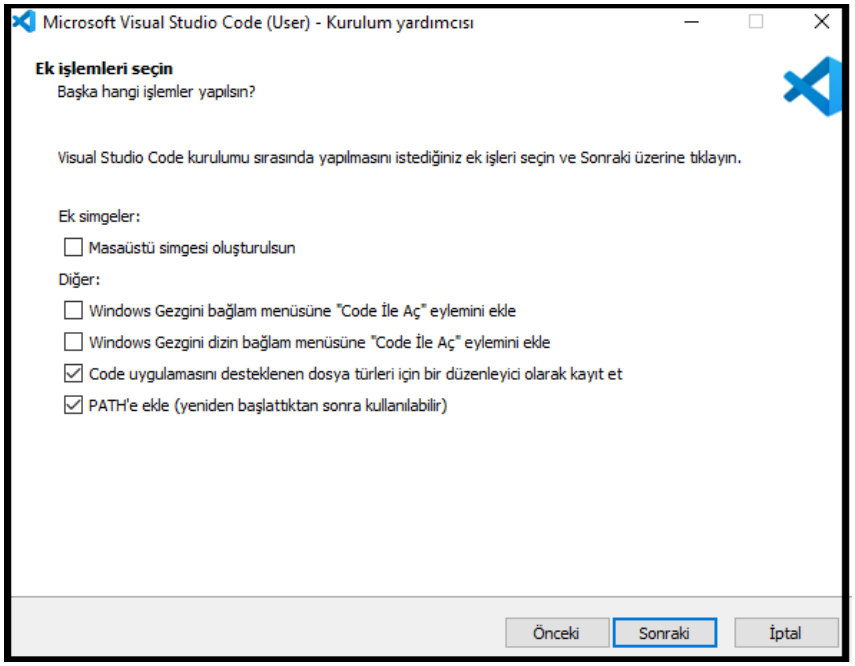
5. VS Code’un yükleneceği konum belirlenmelidir. (Özel bir kurulum dizini ihtiyacı bulunmadığı sürece varsayılan konum yeterli olacaktır.) Akabinde ’İleri’ butonuna tıklanmalıdır.



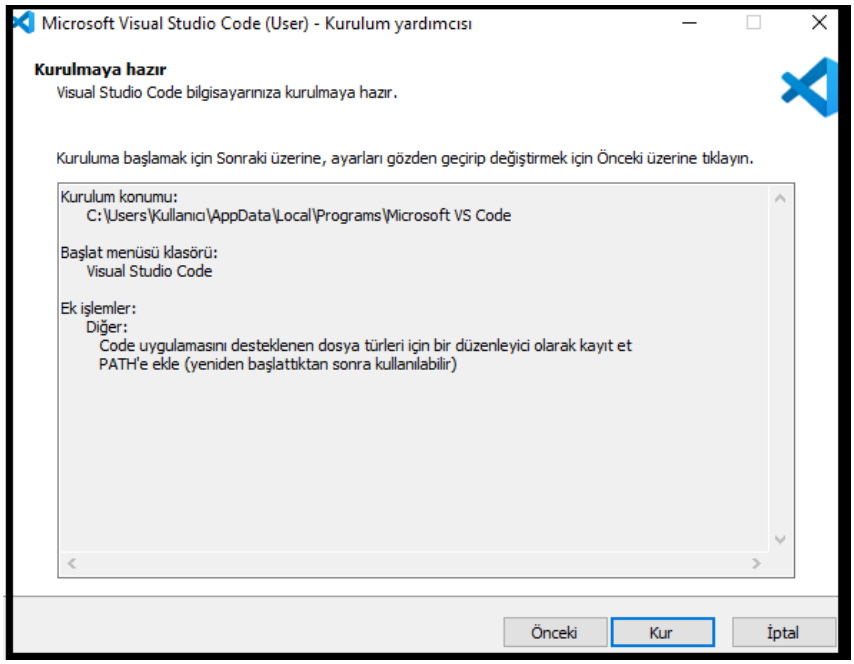
6. VS Code için Başlat Menüsü klasörü belirtilmelidir. (Varsayılan klasörün kullanılması uygun görülmektedir.) Ardından 'İleri' butonuna tıklanarak kuruluma devam edilmelidir.



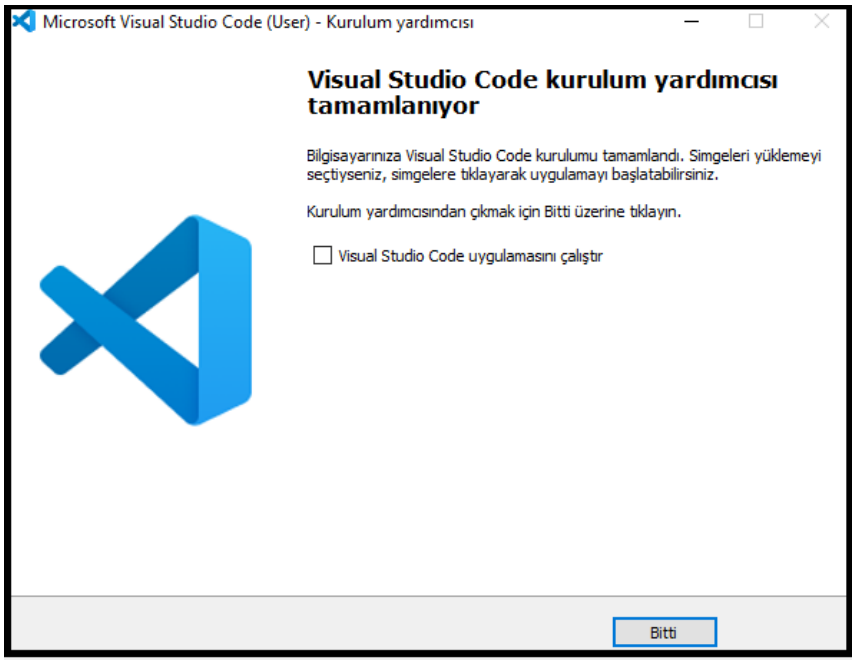
7. İsteğe bağlı ek seçenekler belirlenmelidir. (Varsayılan seçenekler genellikle yeterli olmakla birlikte, masaüstü kısayolu gibi ek bir özellik talep ediliyorsa ilgili seçeneğin işaretlendiğinden emin olunmalıdır.) Akabinde 'İleri' butonuna tıklanmalıdır.



8. Kurulumu başlatmak için ’Kur’ butonuna tıklanmalıdır.



9. Kurulum işlemi başarıyla tamamlanmıştır. Kurulum penceresinden çıkmak için ’Bitti’ butonuna tıklanması gerekmektedir.

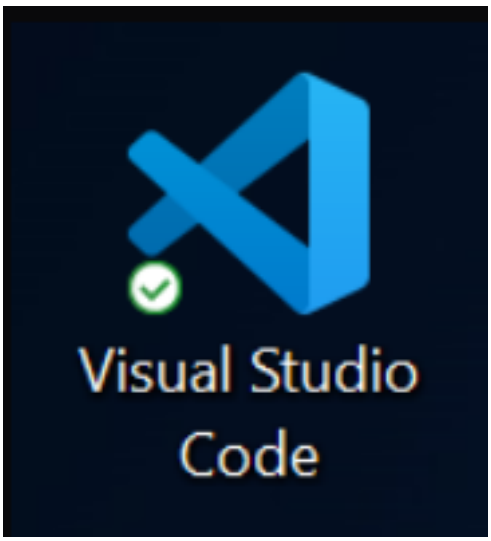


1.6.4 VS Code Çalıştırma

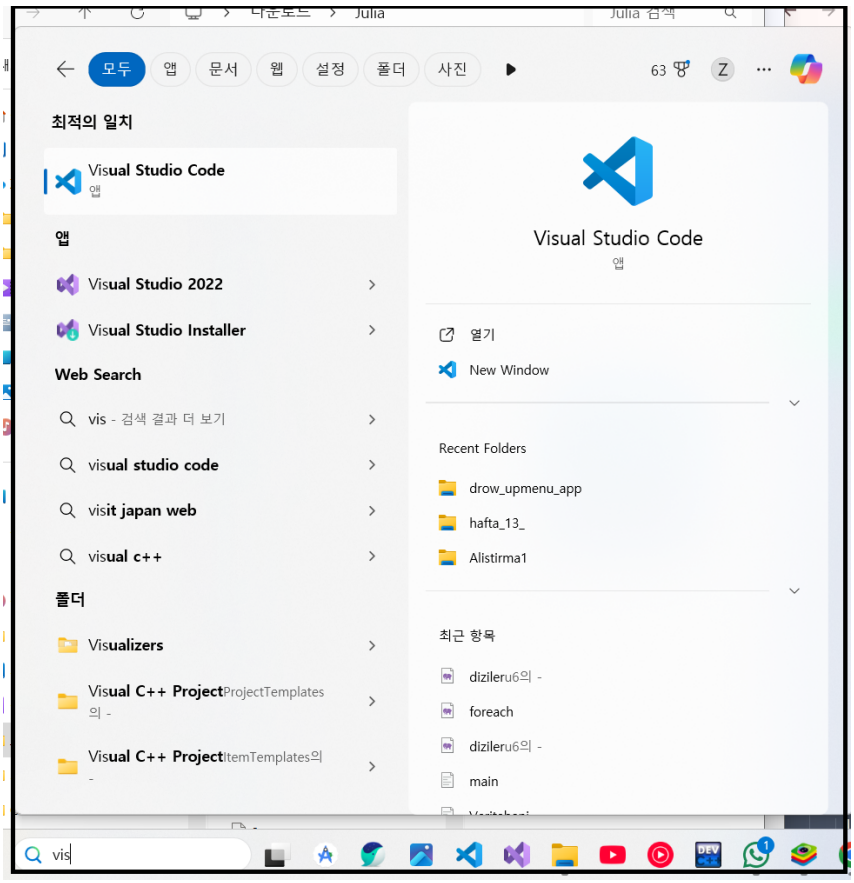
Windows işletim sistemi üzerinde Julia programlama dilinin kurulumu, belirli adımların takip edilmesini gerektirmektedir. Bu bölümde, Julia'nın Windows ortamına kurulum sürecine yönelik detaylı yönergeler sunulacaktır. Kurulum işlemine başlamadan önce, Julia'nın resmi web sitesinden en güncel ve kararlı sürümüne ait yükleyici dosyasının temin edilmesi önemlidir. Elde edilen yükleyici dosyasının çalıştırılması ve kurulum sihirbazının yönlendirmelerine uyulmasıyla kurulum süreci tamamlanacaktır.

Julia'nın yürütülmesi için, yukarıda açıklanan yöntemlerden herhangi biri kullanılarak VS Code çalıştırılabilir:

- Masaüstü kısayolunun kullanılmasıyla.



- Başlat Menüsü'nün kullanılmasıyla.



- Komut satırının kullanılmasıyla.

```
C:\Users\Zeliha>code
```

Bu kılavuzda sunulan adımlar aracılığıyla, Julia programlama dili ve Visual Studio Code entegre geliştirme ortamının Windows işletim sistemi üzerinde başarıyla kurulumu ve çalıştırılması sağlanabilmektedir.

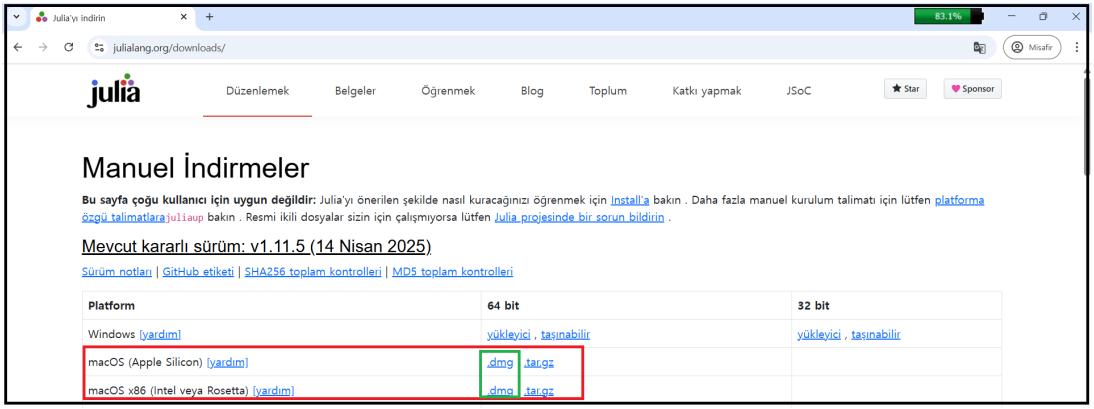
1.7 macOS

Bu bölümde, macOS işletim sistemine sahip kullanıcılar için Julia programlama dili ve entegre geliştirme ortamlarının kurulum süreçleri detaylandırılacaktır. macOS, UNIX tabanlı yapısıyla farklı bir kurulum deneyimi sunmakta olup, hem Julia'nın kendisinin hem de geliştirme ortamı olarak Visual Studio Code'un bu platformda nasıl yapılandırılacağı ele alınacaktır. İşlemci mimarisine (Intel veya Apple Silicon) göre farklılık gösterebilecek kurulum adımları, kullanıcının sorunsuz bir başlangıç yapabilmesi amacıyla ayrıntılı olarak açıklanacaktır.

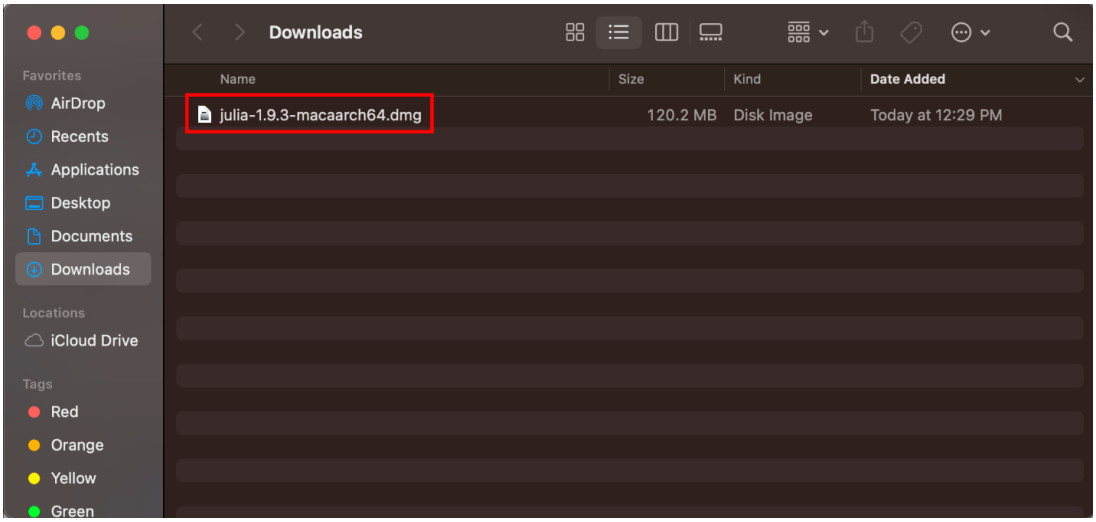
1.7.1 Julia'yı Yükleme

macOS işletim sistemi üzerinde Julia programlama dilinin kurulumu, sistem mimarisine (Apple Silicon veya Intel) bağlı olarak farklılık gösterebilmektedir. Bu bölümde, Julia'nın macOS ortamına kurulum sürecine ilişkin adımlar detaylandırılacaktır. Kurulum işlemi için öncelikle Julia'nın resmi indirme sayfasından ilgili sürüme ait yükleyici dosyasının temin edilmesi gerekmektedir. Ardından, indirilen dosyanın çalıştırılması ve kurulum sihirbazındaki yönergelerin takip edilmesiyle süreç tamamlanacaktır.

1. Julia'nın resmi indirme sayfasına (https://julialang.org/downloads) erişim sağlanmalıdır.
2. 'Güncel kararlı sürüm' başlığı altında yer alan tabloda, 'macOS' etiketiyle belirtilen satır seçilmelidir. Apple Silicon tabanlı sistemler için '64-bit (.dmg)' bağlantısı, Intel tabanlı sistemler veya özel yapılandırma senaryoları için ise Intel veya Rosetta bağlantısı etkinleştirilmelidir.



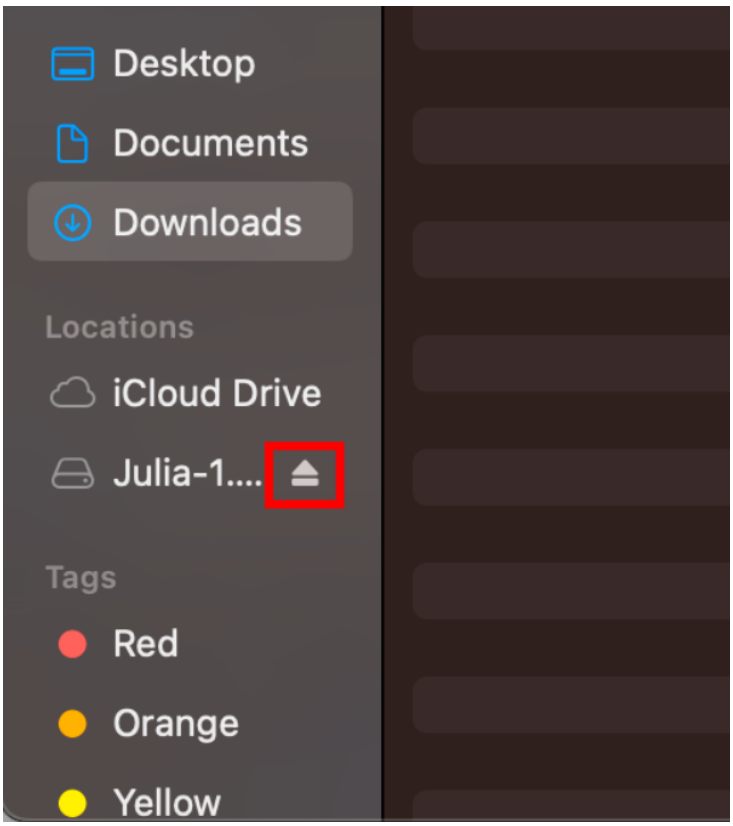
3. İndirilenler klasörü açılmalı ve indirilen ’Julia Disk Image’ dosyasına çift tıklanmalıdır.



4. Açılan pencerede Julia uygulaması, ’Uygulamalar’ klasörüne sürüklenmelidir.

5. Julia'nın kopyalama işlemi için sistemin kimlik doğrulama bilgileri (kullanıcı adı/parola) veya Touch ID onayı sağlanmalıdır.

6. Disk görüntüsü, ilgili 'Çıkar' düğmesi kullanılarak veya bağlam menüsünden ('Sağ Tıkla') 'Çıkart' seçeneği seçilerek sistemden ayrılmalıdır.



7. Julia'nın macOS sistemi üzerinde daha kolay erişilebilir hale getirilmesi amacıyla aşağıdaki talimatların takip edilmesi önerilmektedir.

   Yeni bir terminal penceresi açılmalı ve aşağıdaki komutlar girilmelidir:

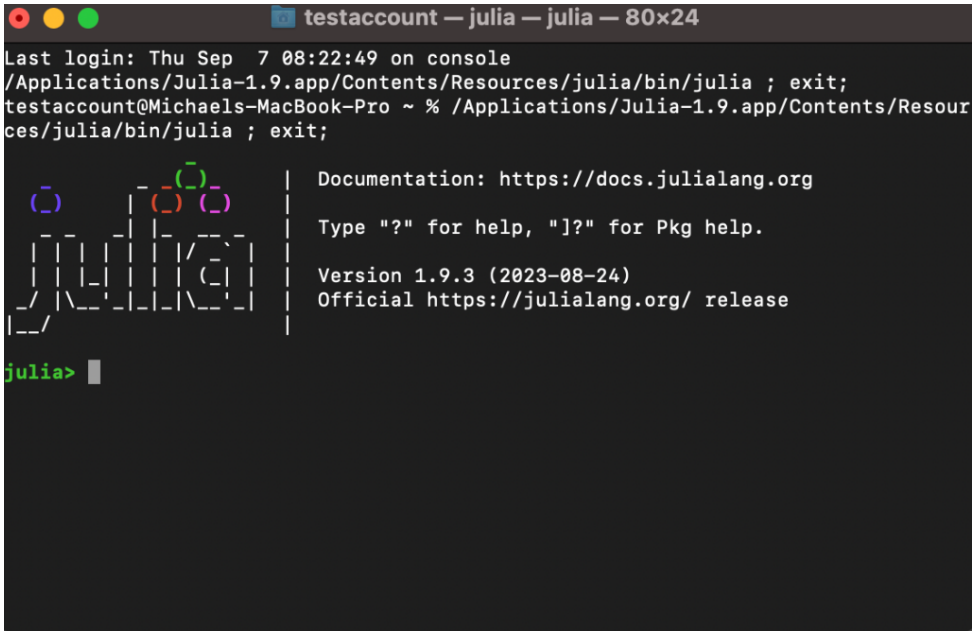
```
sudo mkdir -p /usr/local/bin
sudo rm -f /usr/local/bin/julia
sudo ln -s /Applications/Julia-1.9.app/Contents/Resources/
julia/bin/julia /usr/local/bin/julia
```

1.7.2 Julia'yı çalıştırmak

Julia programlama dilinin macOS ortamında başarıyla kurulumunun ardından, dilin çeşitli yöntemlerle çalıştırılması mümkündür. Bu bölümde, kullanıcıların Julia REPL'ine (Read-Eval-Print Loop) veya Julia kod dosyalarına erişim sağlayabileceği temel çalıştırma yöntemleri açıklanacaktır. Masaüstü kısayolu, Başlat Menüsü (Launchpad) veya komut satırı aracılığıyla Julia'ya erişim imkanları sunulacaktır.

Julia, uygulamalar dizininden standart yöntemlerle başlatılarak veya terminal ortamında (önceki konfigürasyon adımları tamamlandıysa) ilgili komutların girilmesiyle çalıştırılabilir. Ek destek gerektiği takdirde, Apple'ın resmi destek kaynaklarına başvurulabilir.

Julia uygulamasının başlatılmasının ardından, yeni bir Julia komut istemi kullanıcıya sunulacaktır.

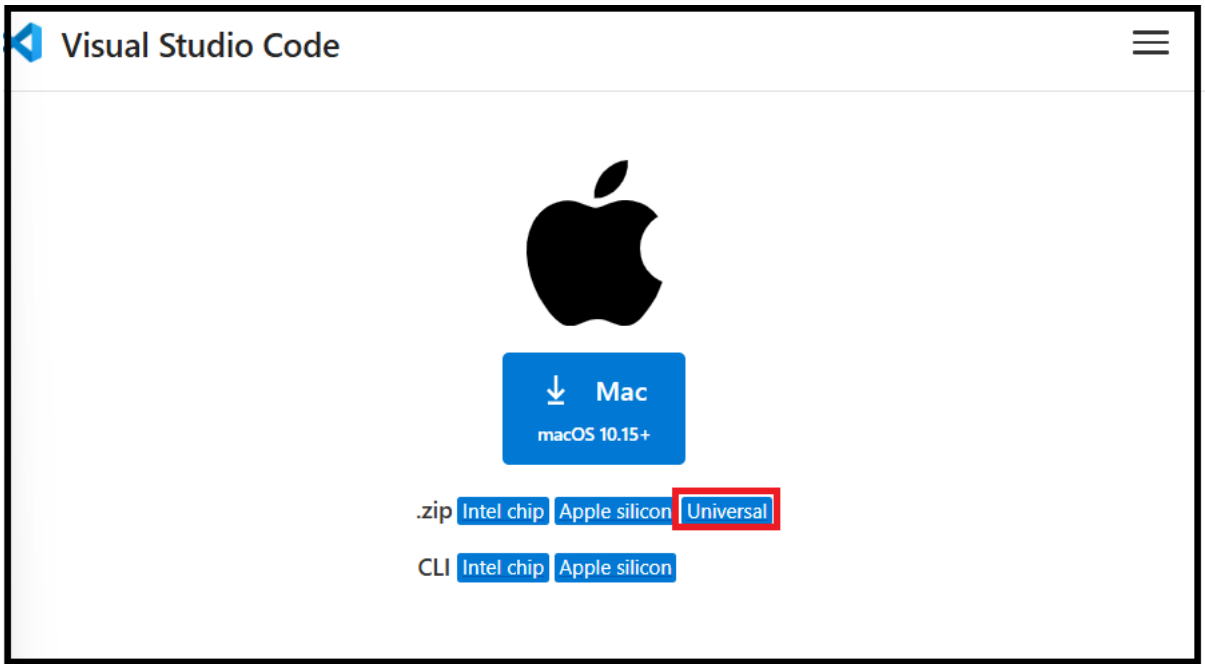


Bu kılavuzda sunulan adımlar aracılığıyla, Julia programlama dilinin macOS işletim sistemi üzerinde başarıyla kurulumu ve çalıştırılması sağlanabilmektedir.

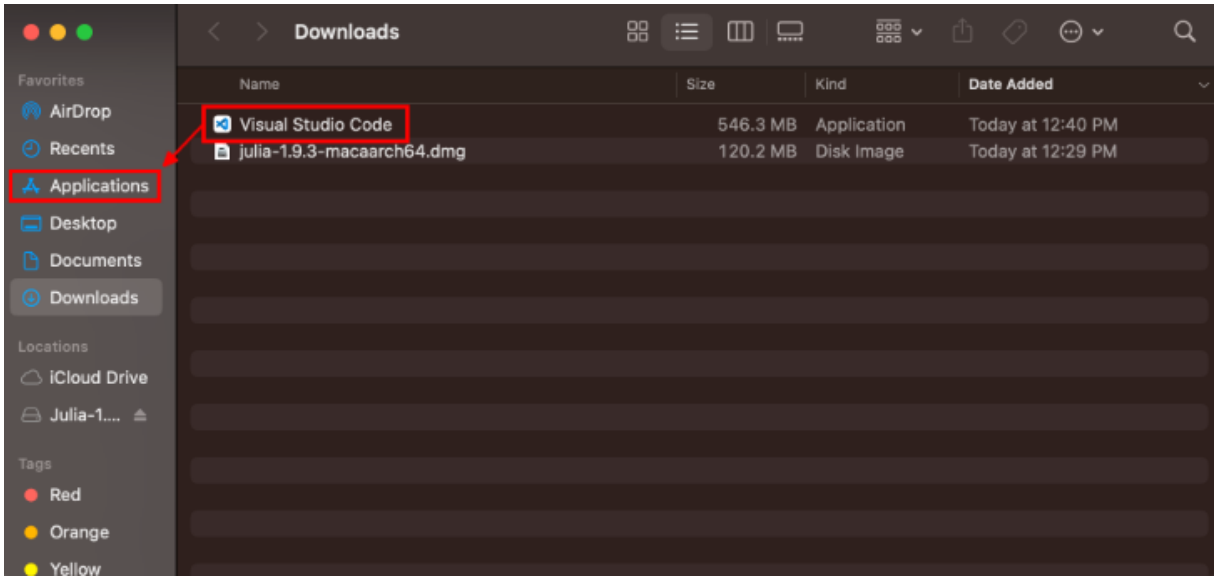
1.7.3 VS Code'u yükleme

macOS işletim sistemi üzerinde Visual Studio Code'un (VS Code) kurulumu, Julia programlama diliyle entegre bir geliştirme ortamı sağlamak için kritik öneme sahiptir. Bu bölümde, VS Code'un macOS ortamına kurulum sürecine yönelik detaylı yönergeler sunulacaktır. Kurulum işlemine başlamadan önce VS Code'un resmi indirme sayfasından güncel yükleyici dosyasının temin edilmesi gerekmektedir. İndirilen dosyanın çalıştırılması ve kurulum sihirbazının yönlendirmelerine uyulmasıyla süreç tamamlanacaktır.

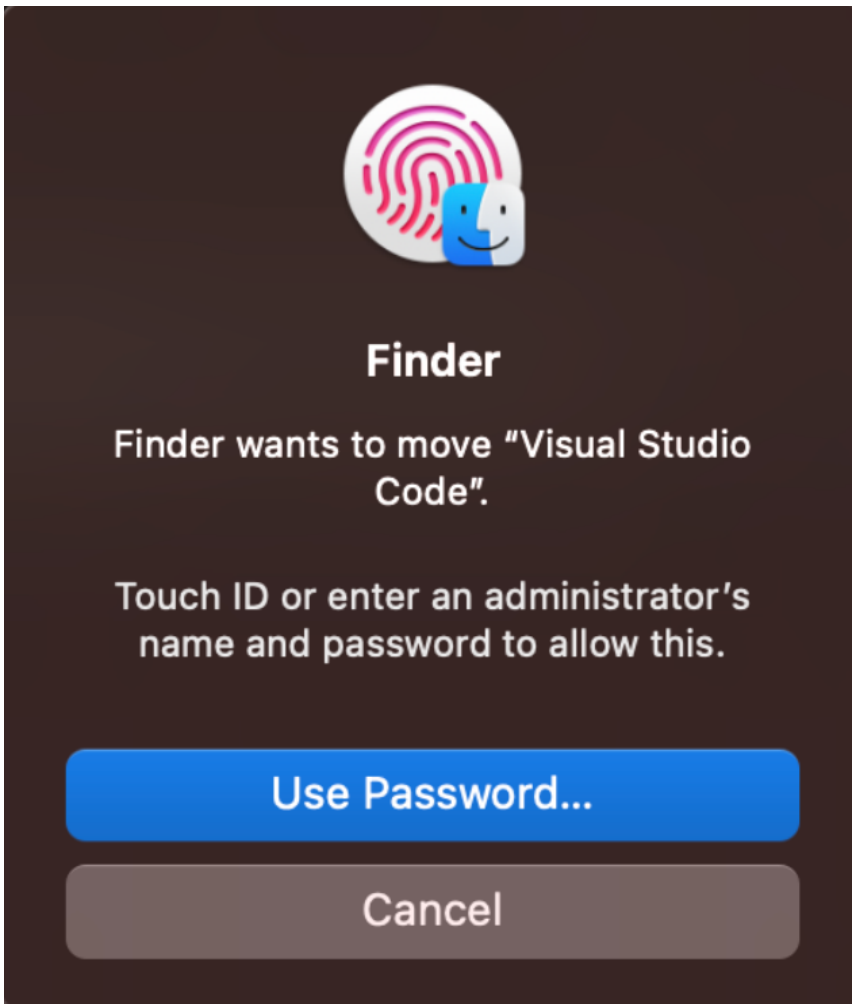
1. Visual Studio Code'un resmi indirme sayfasına (https://code.visualstudio.com) erişim sağlanmalıdır.
2. 'Mac Universal' İndir' butonuna tıklanmalıdır.



3. İndirilenler klasörü açılmalı ve VS Code uygulaması ’Uygulamalar’ klasörüne sürüklenmelidir.



4. VS Code'un taşınma işlemi için sistemin kimlik doğrulama bilgileri (kullanıcı adı/parola) veya Touch ID onayı sağlanmalıdır.



VS Code uygulaması, alışılagelmiş uygulama başlatma yöntemleri aracılığıyla başlatılabilir.

Bu kılavuzda sunulan adımlar aracılığıyla, Visual Studio Code'un macOS işletim sistemi üzerinde başarıyla kurulumu ve çalıştırılması sağlanabilmektedir.

1.8 Linux

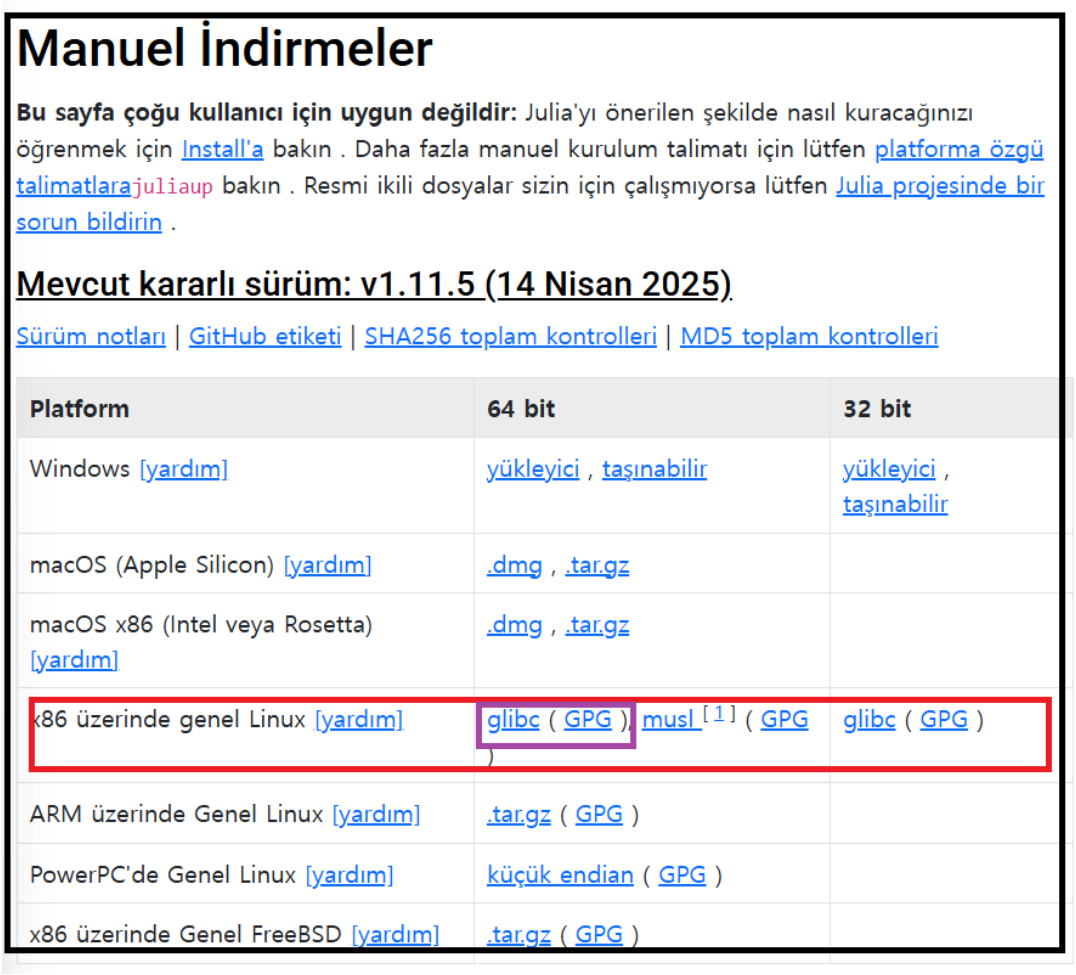
Bu bölümde, Linux işletim sistemi kullanıcıları için Julia programlama dili ve entegre geliştirme ortamlarının kurulum süreçleri detaylandırılacaktır. Açık kaynak kodlu yapısıyla Linux, farklı dağıtımlar ve mimariler üzerinde kurulum esnekliği sunmaktadır. Bu kısım, Julia'nın kendisinin ve ilgili geliştirme ortamlarının Linux platformunda nasıl yapılandırılacağına dair temel adımları ve önemli hususları içerecektir.

1.8.1 Julia'yı Yükleme

Linux işletim sistemi üzerinde Julia programlama dilinin kurulumu, sistem mimarisine ve tercih edilen kurulum yöntemine bağlı olarak değişiklik gösterebilmektedir. Bu bölümde, Julia'nın Linux ortamına kurulum sürecine yönelik detaylı yönergeler sunulacaktır. Kurulum işlemine başlamadan önce, Julia'nın resmi web sitesinden en güncel ve kararlı sürümüne ait yükleyici dosyasının temin edilmesi önemlidir. Elde edilen yükleyici dosyasının çalıştırılması ve kurulum sihirbazının yönlendirmelerine uyulmasıyla veya ilgili komutların terminalde yürütülmesiyle süreç tamamlanacaktır.

1. Julia'nın resmi indirme sayfasına (https://julialang.org/downloads) erişim sağlanmalıdır.

2. Mevcut kararlı sürüm' başlığı altında yer alan tabloda, 'x86 üzerinde Genel Linux' etiketi ile belirtilen satır seçilmelidir. Bu satırdaki '64-bit (glibc)' bağlantısının etkinleştirilmesi gerekmektedir.
   (Mimari farklılıkları bulunan sistemler için uygun alternatif bir indirme bağlantısının seçilmesi gerektiği göz önünde bulundurulmalıdır.)



Manuel İndirmeler

Bu sayfa çoğu kullanıcı için uygun değildir: Julia'yı önerilen şekilde nasıl kuracağınızı öğrenmek için Install'a bakın . Daha fazla manuel kurulum talimatı için lütfen platforma özgü talimatlarajuliaup bakın . Resmi ikili dosyalar sizin için çalışmıyorsa lütfen Julia projesinde bir sorun bildirin .

Mevcut kararlı sürüm: v1.11.5 (14 Nisan 2025)

Sürüm notları | GitHub etiketi | SHA256 toplam kontrolleri | MD5 toplam kontrolleri

Platform	64 bit	32 bit
Windows [yardım]	yükleyici , taşınabilir	yükleyici , taşınabilir
macOS (Apple Silicon) [yardım]	.dmg , .tar.gz	
macOS x86 (Intel veya Rosetta) [yardım]	.dmg , .tar.gz	
x86 üzerinde genel Linux [yardım]	glibc (GPG), musl [1] (GPG)	glibc (GPG)
ARM üzerinde Genel Linux [yardım]	.tar.gz (GPG)	
PowerPC'de Genel Linux [yardım]	küçük endian (GPG)	
x86 üzerinde Genel FreeBSD [yardım]	.tar.gz (GPG)	

3. Bir terminal penceresi açılmalı ve Julia'nın kurulması hedeflenen dizine erişim sağlanmalıdır. Bu dokümanda, Julia'nın /programs/julia/ dizinine kurulumu esas alınmıştır. (Terminal ortamında, '$' işaretinden sonraki ifadeler gerçek komutları temsil etmektedir.)

```
~$ mkdir -p ~/programs/julia
~$ cd ~/programs/julia
```

4. İndirilen arşiv dosyası, belirlenen dizine taşınmalıdır.

```
~/programs/julia$ mv ~/Downloads/julia-1.9.3-linux-x86_64.tar.gz .
```

5. İndirilen arşiv dosyası, belirlenen dizine taşınmalıdır.

```
~/programs/julia$ tar xzf julia-1.9.3-linux-x86_64.tar.gz
```

6. Julia'nın sistem PATH'ine eklenmesi için .bashrc veya ilgili konfigürasyon dosyasına aşağıdaki verilerin eklenmesi gerekmektedir.
   (Kurulum dizininde .zshrc gibi farklı bir kabuk yapılandırma dosyası kullanılıyorsa, ilgili dosyanın güncellenmesi gerektiği göz önünde bulundurulmalıdır.)
   /home/user/programs/julia

```
export PATH="$PATH:/home/user/programs/julia/julia-1.9.3/bin"
```

   (PATH değişkeninin etkin bir şekilde güncellenmesi için terminalin yeniden başlatılması önerilmektedir.)

Bu adımların tamamlanmasıyla Julia programlama dilinin kurulum süreci başarıyla tamamlanmış olur.

1.8.2 Julia'yı çalıştırmak

Julia programlama dilinin Linux ortamında başarıyla kurulumunun ardından, dilin çeşitli yöntemlerle çalıştırılması mümkündür. Bu bölümde, kullanıcıların Julia REPL'ine (Read-Eval-Print Loop) veya Julia kod dosyalarına erişim sağlayabileceği temel çalıştırma yöntemleri açıklanacaktır. Doğrudan yürütülebilir dosya yoluyla veya sistem PATH değişkenine yapılan eklemelerle komut satırından Julia'ya erişim imkanları sunulacaktır.

- Julia, komut satırı aracılığıyla çalıştırılabilir.

```
$ ~/programs/julia/julia-1.9.3/bin/julia
```

- Alternatif olarak, Julia'nın sistem PATH'ine eklenmesi durumunda:

```
$ julia
```

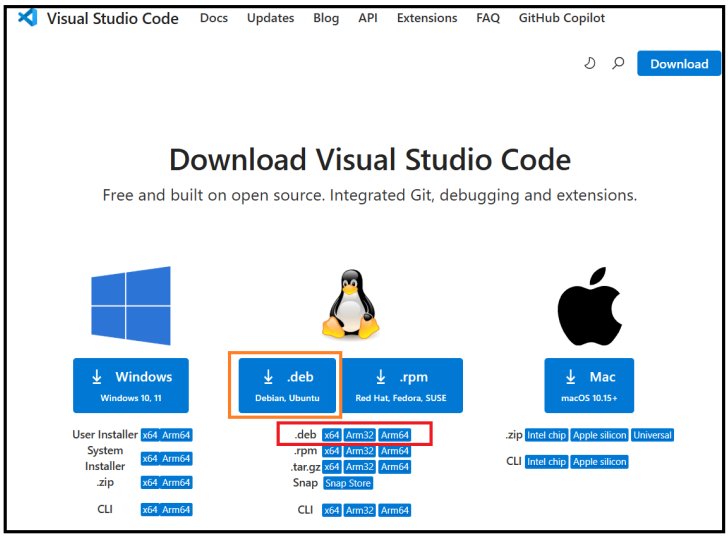
Bu kılavuzda sunulan adımlar aracılığıyla, Julia programlama dilinin Linux işletim sistemi üzerinde başarıyla kurulumu ve çalıştırılması sağlanabilmektedir.

1.8.3 VS Code'u yükleme

Bu bölümde, Linux işletim sistemi üzerinde Visual Studio Code'un (VS Code) kurulum adımları detaylandırılacaktır. VS Code'un kurulumu, geliştirme sürecini kolaylaştıran çeşitli özelliklere erişim sağlamak için kritik öneme sahiptir. Sunulan adımlar özellikle Ubuntu dağıtımı üzerinde test edilmiş olup, Debian tabanlı diğer dağıtımlarda da genellikle benzer şekilde uygulanabilir. Ancak, farklı Linux dağıtımları için bazı spesifik uyarlamalar gerekebileceği dikkate alınmalıdır. Kurulum işlemine başlamadan önce VS Code'un resmi indirme sayfasından güncel yükleyici dosyasının temin edilmesi önemlidir.

Bu bölümde sunulan kurulum adımları, Ubuntu işletim sistemi yüklü bir sistem üzerinde test edilmiştir. Söz konusu adımların, Debian tabanlı diğer Linux dağıtımlarında da uygulanabilir olduğu varsayılmakla birlikte, farklı Linux dağıtımlarında ek uyarlamalar gerekebileceği belirtilmelidir. Kurulum adımlarından bazıları için yönetici ayrıcalıklarının (root/sudo) gerekliliği göz önünde bulundurulmalıdır.

1. Visual Studio Code'un resmi indirme sayfasına (https://code.visualstudio.com) erişim sağlanmalıdır.
2. '.deb' uzantılı kurulum dosyasına tıklanmalıdır.



3. (Dosya yöneticisi açılarak indirilen dosya çift tıklama yoluyla etkinleştirilmeli ya da VS Code, komut satırı üzerinden yüklenmelidir. (Gerektiği durumlarda dosya yolu veya adının güncellenmesi gerektiği belirtilmelidir.)

```
$ sudo dpkg -i /home/user/Downloads/code_1.81.1-1691620686_amd64.deb
```

Visual Studio Code kurulumu başarıyla tamamlanmıştır.

1.8.4 VS Code Çalıştırma

Visual Studio Code'un Linux ortamında başarıyla kurulumunun ardından, IDE'nin çeşitli yöntemlerle başlatılması mümkündür. Bu bölümde, kullanıcıların VS Code arayüzüne veya entegre terminale erişim sağlayabileceği temel çalıştırma yöntemleri açıklanacaktır. Masaüstü kısayolu, uygulama menüsü veya komut satırı aracılığıyla VS Code'a erişim imkanları sunulacaktır.

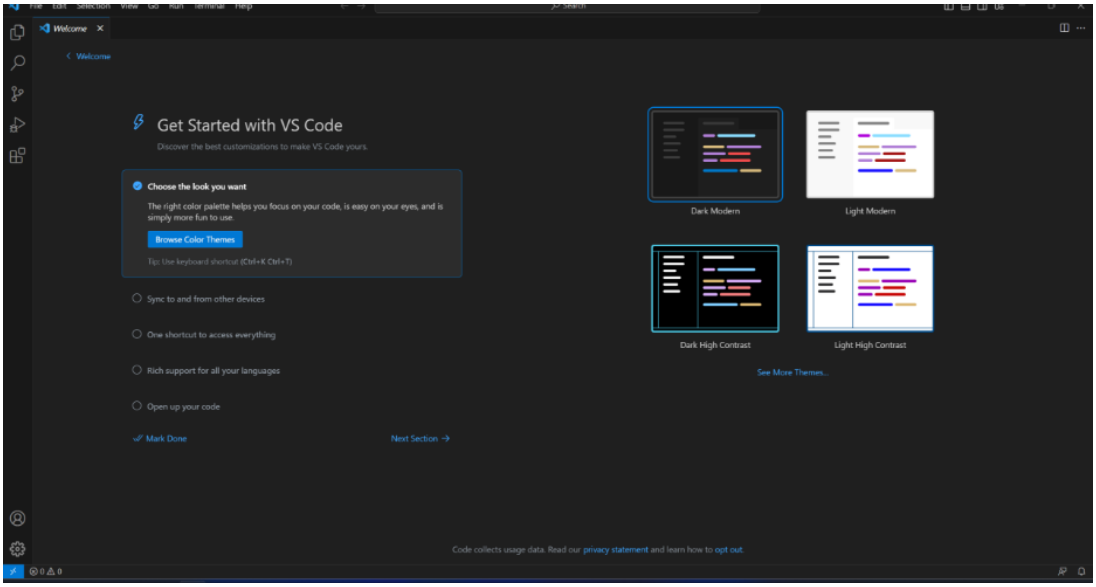
```
$ code
```

VS Code, komut satırı aracılığıyla çalıştırılabilir. Bu kılavuzda sunulan adımlar aracılığıyla, Visual Studio Code'un Linux işletim sistemi üzerinde başarıyla kurulumu ve çalıştırılması sağlanabilmektedir.

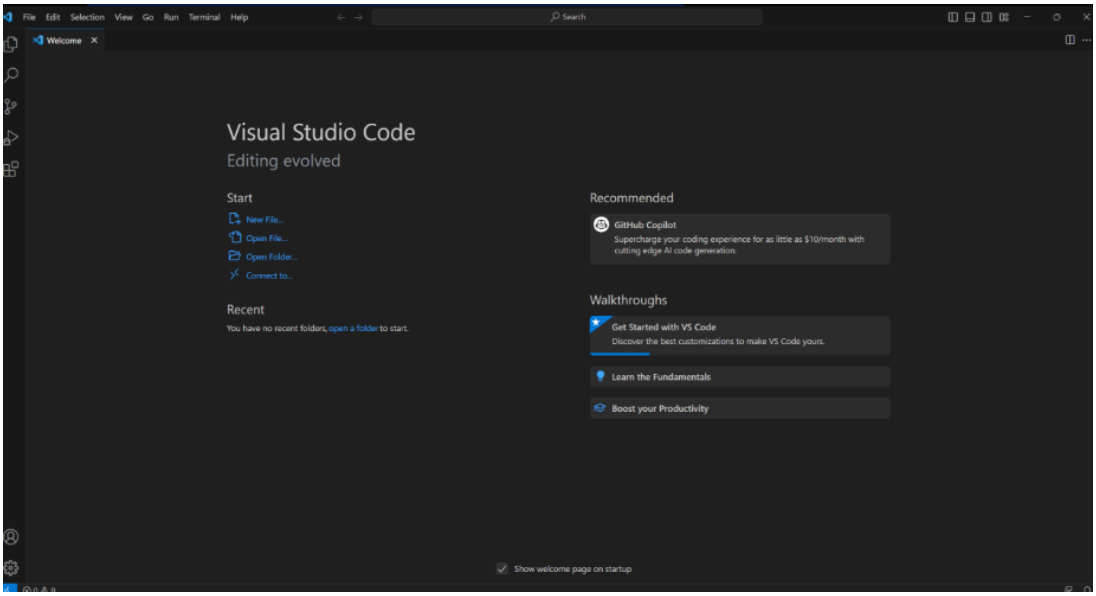
1.9 Julia için VS Code'u yapılandırma

Visual Studio Code'un Julia programlama diliyle etkin bir şekilde entegre olabilmesi için gerekli yapılandırma adımları bu bölümde açıklanacaktır. Bu entegrasyonun temelini, Julia diline özel işlevsellikleri sağlayan Julia eklentisinin kurulması oluşturmaktadır. Eklenti kurulumunun ardından, Visual Studio Code'un Julia yürütülebilir dosyasının konumunu doğru bir şekilde algılaması için gerekli yapılandırma süreçleri de bu bölümde detaylandırılacaktır. Bu adımlar, Julia geliştirme ortamının kusursuz bir şekilde işlev görmesini temin edecektir

Visual Studio Code'un ilk kez başlatılması durumunda, 'VS Code ile Başlayın' mesajı görüntülenecektir.

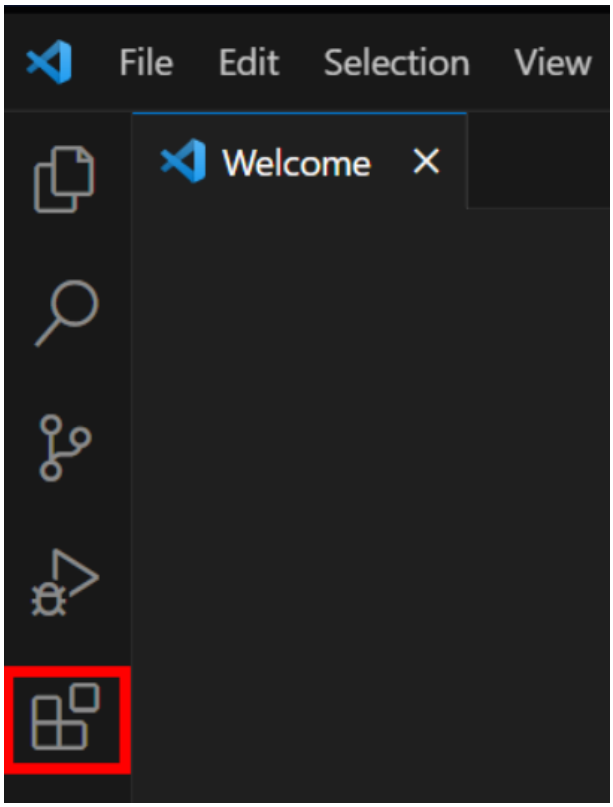


Kullanıcılar, sunulan başlangıç seçeneklerini inceleyebilirler; ancak bu kılavuzun kapsamı dahilinde, başlangıç ekranındaki diğer seçenekler göz ardı edilerek pencerenin sol üst köşesinde yer alan 'Hoş Geldiniz' ibaresine tıklanmalıdır. Bu işlem, ilgili hoş geldiniz sayfasına yönlendirme sağlayacaktır.

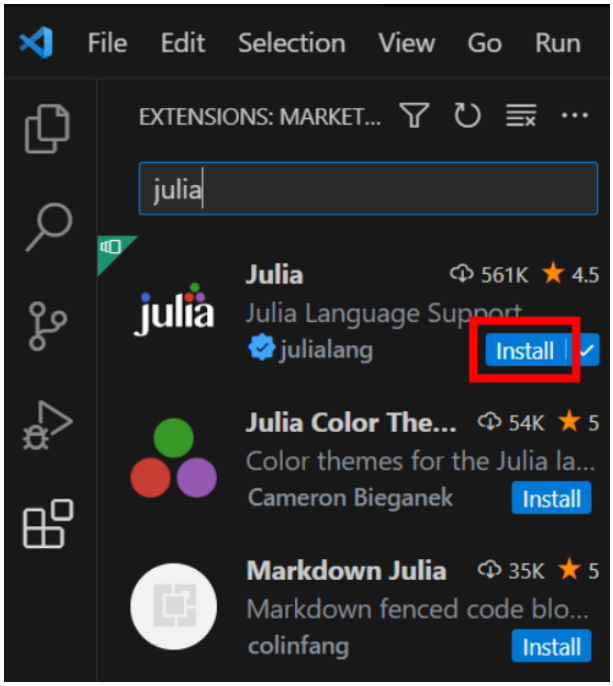


Daha sonra, Visual Studio Code için Julia eklentisinin kurulumu gerekmektedir.

1. Uzantılar butonuna tıklanmalıdır..

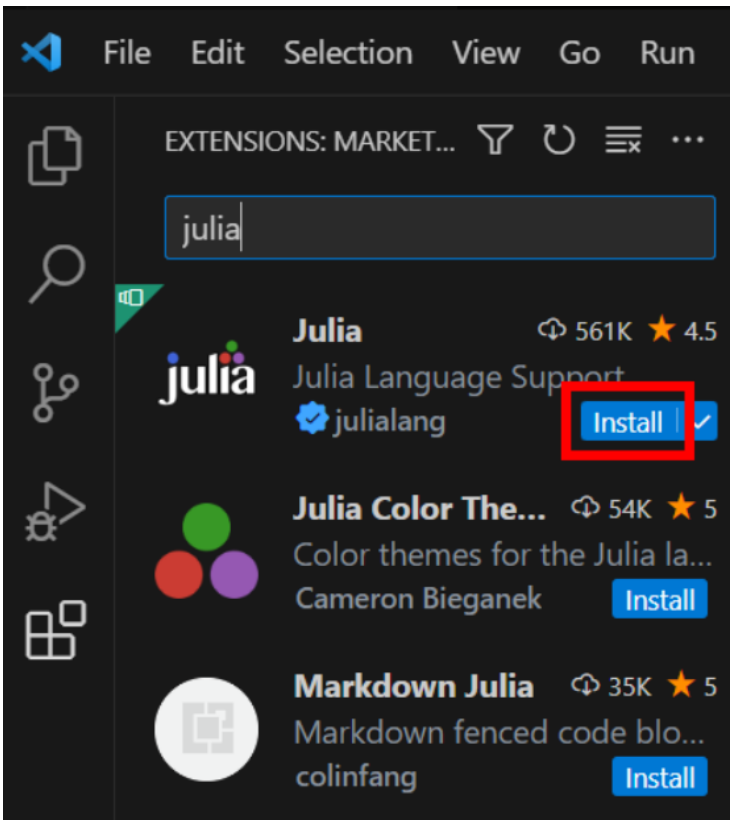


2. 'Julia' araması yapılmalı ve Julia eklentisi bulunduğunda 'Yükle' seçeneği etkinleştirilmelidir.

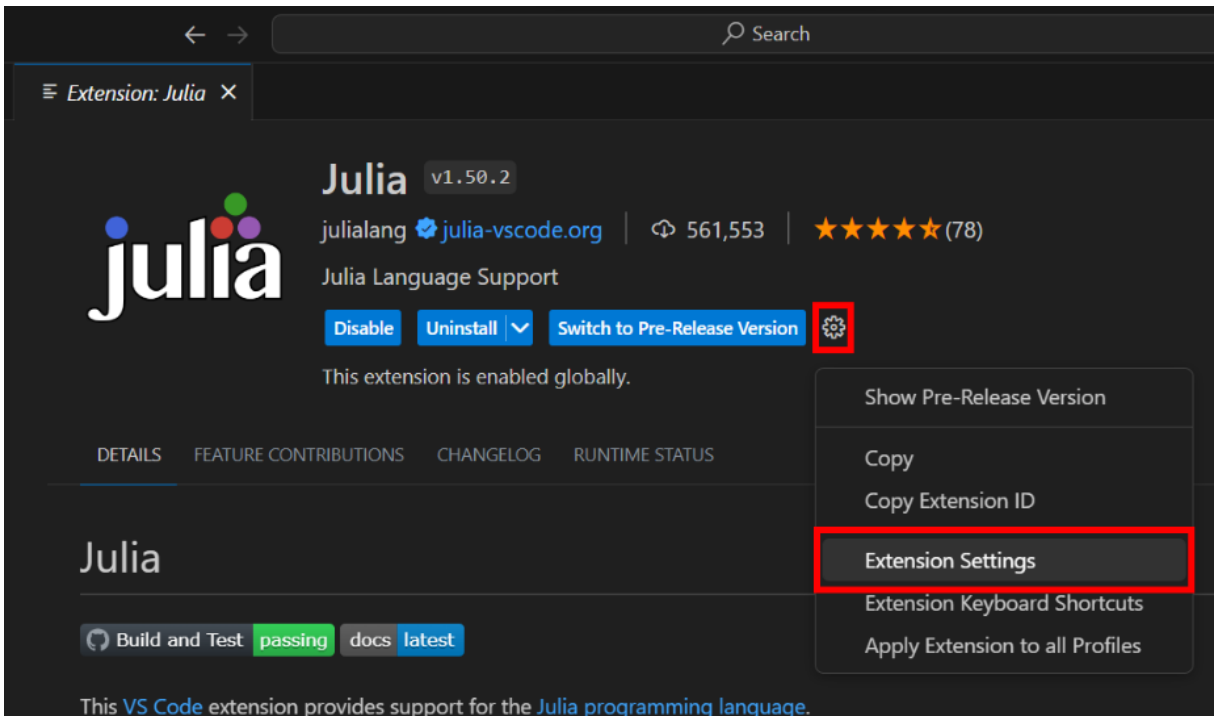


Julia eklentisinin, Julia programlama dilinin kurulum konumunu doğru bir şekilde algıladığından emin olunması gerekmektedir. Julia'nın sistem PATH değişkenine eklenmiş olması durumunda, bu yapılandırma adımı atlanabilir. Ancak, Visual Studio Code içerisinde Julia'yı çalıştırma sırasında hata oluşması veya yanlış Julia sürümünün kullanıldığı tespit edilmesi durumunda (özellikle birden fazla Julia sürümü yüklüyse), aşağıdaki adımların takip edilmesi önerilmektedir.

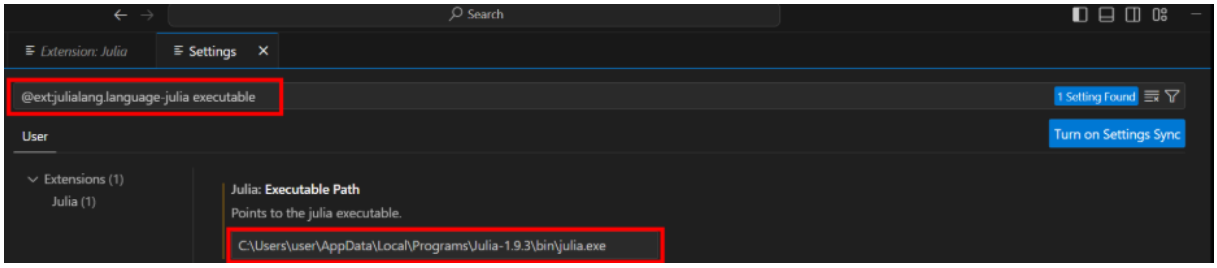
1. Ayarlar simgesi kullanılarak Julia eklenti ayarları açılmalıdır. Ardından, 'Eklenti Ayarları' seçeneği tıklanmalıdır.



1. Ayarlar simgesi kullanılarak Julia eklenti ayarları açılmalıdır. Ardından, 'Eklenti Ayarları' seçeneği tıklanmalıdır.



2. 'Yürütülebilir yol' araması yapılmalıdır. Ardından, 'Julia: Yürütülebilir Yol' uzantı ayarında Julia yürütülebilir dosyasının yolu belirtilmelidir.

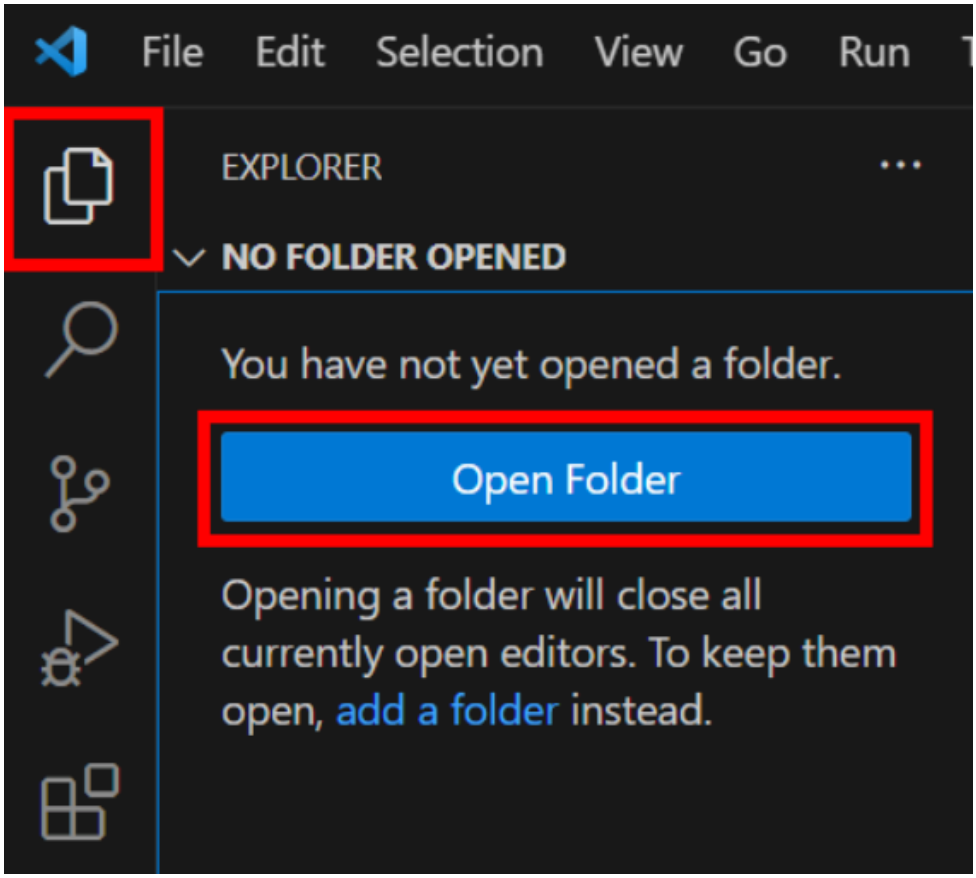


Bu adımların tamamlanmasıyla Visual Studio Code, Julia geliştirme ortamı için hazır hale gelmiştir.

1.10 VS Code'da Julia Kodunu Geliştirme

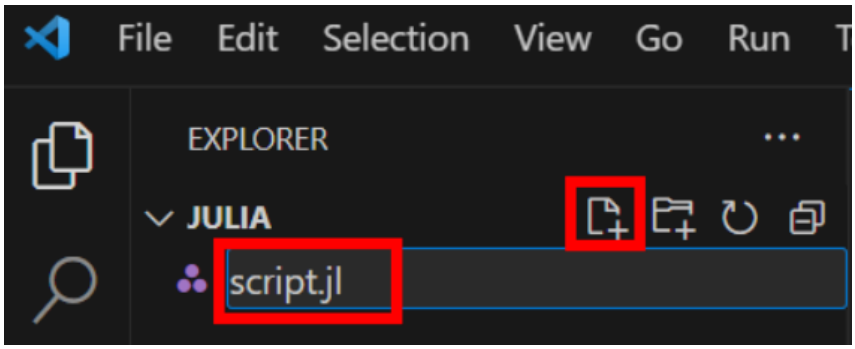
Julia programlama dilini Visual Studio Code ortamında etkin bir şekilde kullanabilmek için yeni projelerin ve kod dosyalarının oluşturulması temel bir adımdır. Bu bölümde, VS Code arayüzü içerisinde Julia kodunu yazma, kaydetme ve çalıştırma süreçleri adım adım açıklanacaktır. Kod geliştirme pratiğine yönelik bu yönergeler, kullanıcıların projelerini verimli bir şekilde yönetmelerine olanak tanıyacaktır.

1. Explorer bölmesini açmak amacıyla sol üst köşede yer alan dosya simgesine tıklanmalıdır.Ardından, 'Klasörü Aç' seçeneği tıklanmalıdır.



2. Julia kodunun kaydedileceği bir dizin seçilmeli veya yeni bir dizin oluşturulmalıdır. Bu örnekte, '**Julia**' adında bir dizin oluşturulmuştur.

3. Explorer bölmesinde ilgili klasörün yanında bulunan 'Yeni Dosya' düğmesine tıklanmalıdır. Sonrasında, Julia kodunun yazılacağı dosya için bir adlandırma yapılmalı ve dosyanın '**.jl**' uzantısına sahip olduğundan emin olunmalıdır. Bu kılavuzda, '**script.jl**' adında bir dosya oluşturulmuştur.

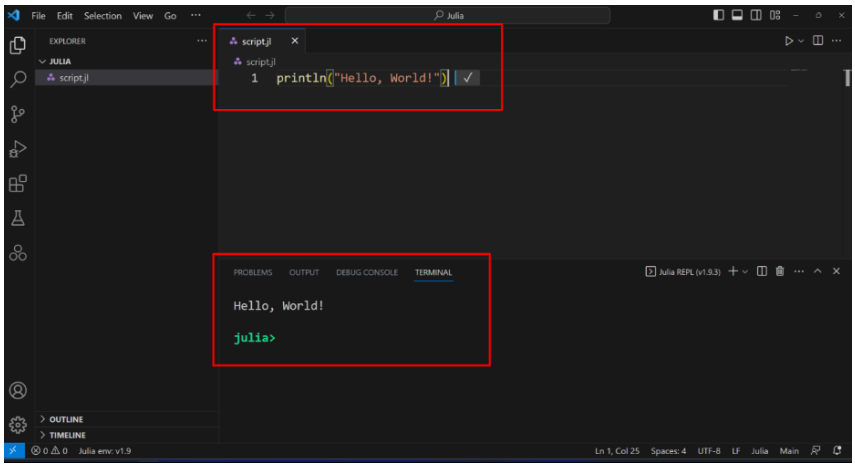


Visual Studio Code, düzenleme için hazır boş bir dosya içeren yeni bir sekme açacaktır.

4. Aşağıdaki kodun oluşturulan dosyaya eklenmesi ve kaydedilmesi gerekmektedir.

```
println("Hello, World!")
```

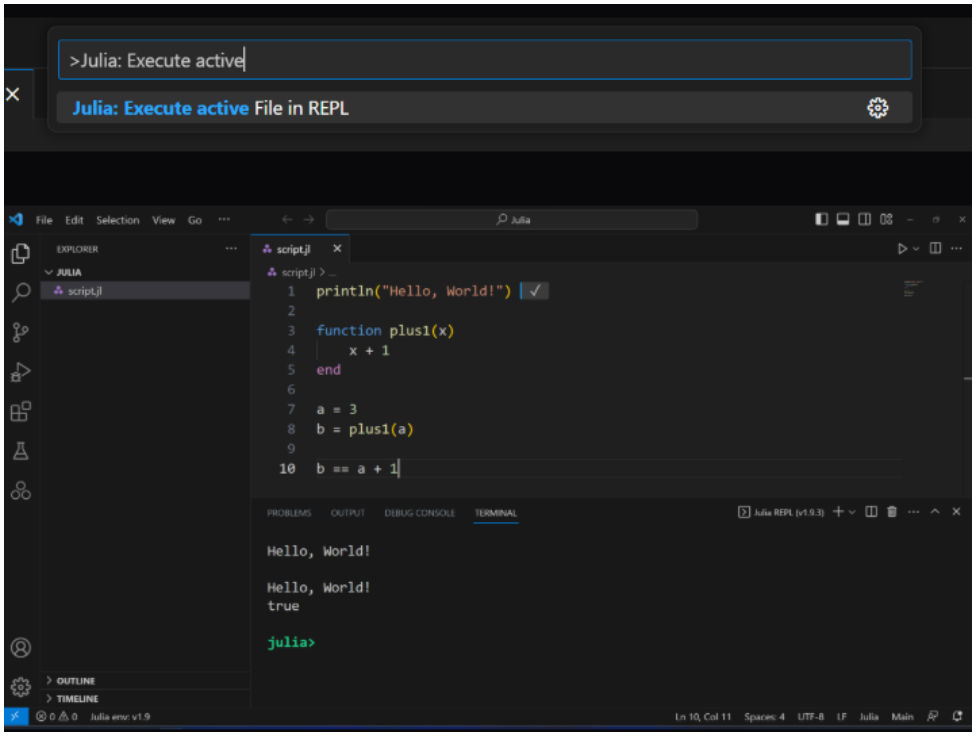
5. Kodu çalıştırmak amacıyla imleç, yürütülecek kod satırına konumlandırılmalı ve ardından '**Shift+Enter**' tuş kombinasyonuna basılmalıdır. Bu eylem, daha önce başlatılmamışsa bir Julia oturumu başlatacak ve ardından kodu çalıştıracaktır. Kodun ilk kez çalıştırılması durumunda, paketlerin ön derleme ihtiyacından dolayı bir miktar zaman alabileceği göz önünde bulundurulmalıdır.



6. Ek kod satırları aşağıda belirtildiği gibi eklenebilir.

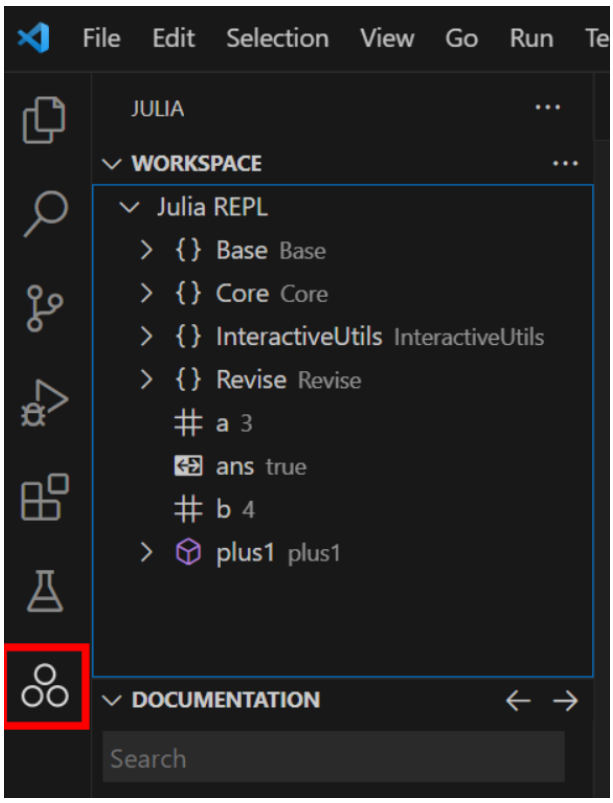
```
function plus1(x)
    x + 1
end

a = 3
b = plus1(a)

b == a + 1
```

7. D Dosyadaki tüm kodun çalıştırılması için komut paleti (’**Ctrl+Shift+P**’) açılmalı ve ’Julia: Execute active file in REPL’ ifadesi aranmalıdır. İlgili komut vurgulandıktan sonra ’Enter’ tuşuna basılmalıdır.

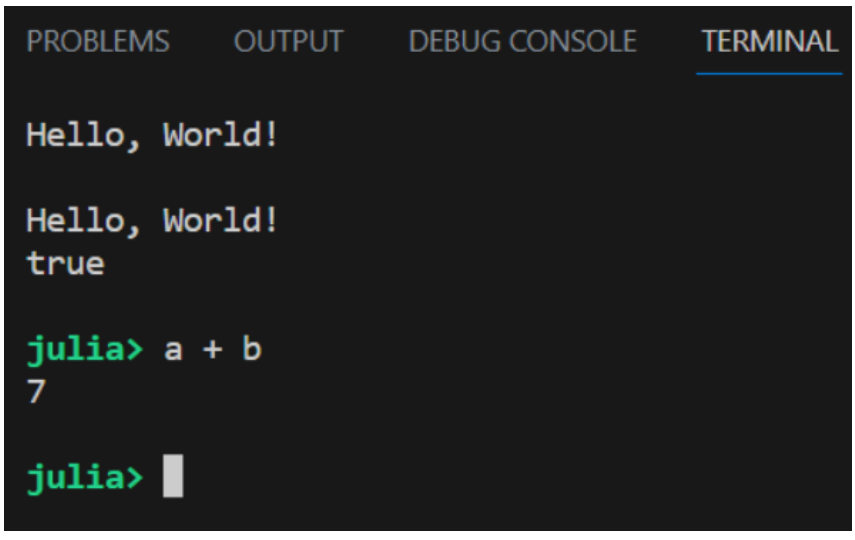


REPL’e iki satır çıktının eklendiği gözlemlenecektir: ifade değeri ve yürütülen son satırın sonucu (örn. print(değer), b == a + 1 gibi).

8. Kodun çalıştırılmasının ardından, sol kısımda yer alan Julia bölmesi incelenerek 'Çalışma Alanı' sekmesinde tanımlanmış değişkenler ve fonksiyonlar görüntülenebilir.



9. REPL (Read-Eval-Print Loop) doğrudan kullanılabilir.



Bu adımlarla birlikte, Visual Studio Code ortamında Julia kodunun nasıl yazılacağı ve çalıştırılacağı öğrenilmiş olmaktadır.

2 -Söz Dizimi(Syntax)

Her programlama dilinde olduğu gibi, Julia'nın da kendine özgü bir söz dizimi yapısı bulunmaktadır. Bu bölümde, Julia programlama dilinin temel söz dizimi kuralları detaylandırılacaktır. Yorum satırlarından başlayarak, değişken bildirimleri, operatör kullanımları ve temel kontrol yapıları gibi dilin yapı taşları incelenecektir. Bu kuralların anlaşılması, Julia ile etkin kod yazma yeteneğinin geliştirilmesi için kritik öneme sahiptir.

2.1 Julia'ya Giriş: Dinamik Tiplendirme ve Tam Zamanında Derleme

Julia, dinamik olarak tiplendirilmiş ve Tam Zamanında (Just-In-Time - JIT) derleyiciye sahip yüksek seviyeli bir programlama dilidir. Bu yapı, C++ veya FORTRAN gibi statik olarak derlenmiş dillerde geleneksel olarak gerekli olan belirgin bir ön derleme adımını ortadan kaldırır. Bunun yerine, Julia kaynak kodu alır, ihtiyaç duyulduğunda tip çıkarımı (type inference) yapar ve kodun ilgili bölümlerini çalıştırma anından hemen önce derler. Bu yaklaşım, hem dinamik dillerin esnekliğini hem de derlenmiş dillerin yüksek performansını bir araya getirmeyi hedefler. Ayrıca, her değişken veya ifadenin tipini açıkça belirtme zorunluluğu yoktur; Julia, tipleri sizin için otomatik olarak çıkarır.

Julia'nın R ve Python gibi diğer popüler dinamik programlama dillerinden temel ayrımları şunlardır:

İsteğe Bağlı Tip Notasyonu: İlk olarak, Julia, kullanıcıların isteğe bağlı tip notasyonu kullanmasına olanak tanır. Bu notasyon, genellikle değişken isimlerinden veya fonksiyon parametrelerinden sonra gelen '::' operatörü ile ifade edilir. Bu, geliştiricinin performans veya bellek kullanımı gibi hususları optimize etmek amacıyla tipler hakkında derleyiciye ek bilgi sağlamasına olanak tanır. Ancak, bu tip bildirimleri zorunlu değildir; Julia, gerektiğinde tipleri otomatik olarak çıkaracaktır.

Çoklu Dağıtım (Multiple Dispatch): İkinci olarak, Julia'nın çekirdek özelliklerinden biri Çoklu Dağıtım mekanizmasıdır. Bu mekanizma, aynı fonksiyon adının, çağrıldığı argümanların tiplerine göre farklı davranışlar sergilemesine olanak tanır. Geliştiriciler, aynı fonksiyon adı altında, farklı argüman tipi kombinasyonları için yeni fonksiyon imzaları (signatures) tanımlayarak tip tabanlı özelleştirilmiş davranışlar belirleyebilirler. Bu özellik, özellikle sayısal ve bilimsel hesaplamalar bağlamında esneklik ve modülerlik sağlar.

2.2 Değişkenler

Değişkenler, bir bilgisayarda belirli bir adla depolanan ve daha sonra geri çağrılabilen veya değiştirilebilen değerlerdir. Julia, farklı değişken türlerini desteklemekle birlikte, veri biliminde başlıca aşağıdaki türler kullanılmaktadır:

- Tamsayılar:**Int64**
- Gerçek Sayılar:**Float64**
- Mantıksal Değerler**Bool**
- Karakter Dizileri:**String**

Tam sayılar ve gerçek sayılar, varsayılan olarak 64 bit kullanılarak saklanmaktadır; bu nedenle ilgili tiplerin adlarında **64** son eki bulunmaktadır. Daha yüksek veya daha düşük kesinlik ihtiyacı durumunda, örneğin 8 bitlik 'Int8' veya 128 bitlik **Int128** gibi farklı bit boyutlarında veri tipleri mevcuttur. Bu tiplerin ihtiyaca göre kullanılması önerilmektedir.

Değişkenler, adları sola ve değerleri sağa yazılarak, ortada '=' atama operatörü kullanılarak oluşturulmaktadır. Örneğin:

```
name = "Julia"
age = 9
```

Dönüş Çıktısı:

```
9
```

Son ifadenin (**age**), dönüş çıktısının konsola yazdırıldığı unutulmamalıdır. Bu örnekte **name** ve **age** olmak üzere iki yeni değişken tanımlanmıştır. Tanımlanan bu değişkenlerin değerleri, atamada belirtilen adlar kullanılarak çağrılabilir.

```
name
```

Dönüş Çıktısı:

```
Julia
```

Mevcut bir değişken için yeni değerler tanımlanmak istendiğinde, atama adımları tekrarlanabilir. Bu durumda, Julia'nın önceki değeri yeni değerle geçersiz kılacağı göz önünde bulundurulmalıdır. Aşağıdaki örnekte, 'age' değişkeninin değeri güncellenmiştir:

```
age = 10
```

Dönüş Çıktısı:

```
10
```

'**name**' değişkeni için de benzer bir işlem gerçekleştirilebilir. Değişken, yeni değeri 'Julia Rapidus' olarak güncellenmiştir:

```
name = "Julia Rapidus"
```

Dönüş Çıktısı:

```
Julia Rapidus
```

Değişkenler üzerinde toplama veya bölme gibi işlemler de gerçekleştirilebilir. Örneğin, Julia'nın yaşı 12 ile çarpılarak ay cinsinden değeri hesaplanabilir:

```
12 * age
```

Dönüş Çıktısı:

```
120
```

Değişkenlerin tipleri aşağıdaki fonksiyon kullanılarak incelenebilir:

```
typeof(age)
```

Dönüş Çıktısı:

```
Int64
```

3 -Operatörler

Julia programlama dili, temel sayısal veri tipleri üzerinde standart aritmetik ve karşılaştırma operatörlerinin yanı sıra kapsamlı bir matematiksel operatör seti sunmaktadır. Bu bölümde, söz konusu temel aritmetik ve mantıksal operatörlere genel bir bakış sağlanacaktır. Burada sunulan bilgiler giriş niteliğinde olup, belirli operatörlerin ve kullanımlarının daha detaylı incelenmesi ilerleyen bölümlere bırakılacaktır. Bu nedenle, bazı noktaların bu aşamada tam olarak anlaşılamaması durumunda endişe duyulmamalı; ilgili konular ilerleyen bölümlerde daha kapsamlı bir şekilde ele alınacaktır.

3.1 Aritmetik Operatörler

Takip eden kısımda incelenecek olan temel aritmetik operatörler, Julia'nın temel sayısal veri türleri tarafından desteklenmektedir.

İfade	İsim	Tanım
`+x`	tekli artı	sayının işaretini değiştirmez, olduğu gibi bırakır.
`−x`	tekli eksi	bir sayının işaretini tersine çevirir
`x + y`	ikili artı	iki sayıyı (veya değişkeni) toplar
`x − y`	ikili eksi	ilk sayıdan ikinci sayıyı çıkarır
`x * y`	çarpma	iki sayıyı çarpar
`x / y`	bölme	ilk sayıyı ikinci sayıya böler (kayan noktalı sonuç)
`x ÷ y`	tam sayı bölme	x'in y'ye bölümünün tam sayı kısmını verir
`x \y`	ters bölme	ikinci sayıyı ilk sayıya böler (y/x)
`x ^y`	üs alma	x sayısının y. kuvvetini hesaplar
`x % y`	mod alma	x'in y'ye bölümünden kalanı verir

Tablo 1: Julia'daki temel aritmetik operatörler.

+x Tekli Artı (Unary Plus): Etkisiz bir işlem olarak düşünülebilir. Bir sayının önüne geldiğinde sayının işaretini değiştirmez, olduğu gibi bırakır.

Örnek olarak: `x = 5` durumunda `+x` değeri yine `5` olarak kalır; benzer şekilde `y = -3` durumunda `+y` değeri yine `-3` olacaktır. Çoğu zaman doğrudan bir etkisi olmamasına rağmen, bu operatör bazen kodun okunabilirliğini artırmak veya metotların (fonksiyonların) tutarlılığını sağlamak amacıyla kullanılmaktadır.

Örneğin: `(-)`Tekli eksi (Unary Minus) operatörü mevcut olduğunda, buna simetrik olarak `(+)` tekli artı (Unary Plus) operatörü de bulunmaktadır.

-x Tekli eksi (Unary Minus): Değerleri toplamsal tersine eşleyen bu operatör, sayının işaretini tersine çevirme işlevi görür. Pozitif bir değeri negatif, negatif bir değeri ise pozitif hale getirir.

Örnek: `x = 5` ise `-x` sonucu -5 olur. `y = -3` ise `-y` sonucu 3 olur.

x + y İkili Artı (Binary Plus): Toplama işlemi yapar. İki sayıyı (veya değişkeni) toplar.

Örnek: `3 + 4` sonucu 7'dir.

x - y İkili Eksi (Binary Minus): Çıkarma işlemi yapar. İlk sayıdan ikinci sayıyı çıkarır.

Örnek: `7 - 2` sonucu 5'tir.

x * y Çarpma(Times): Çarpma işlemi yapar. İki sayıyı çarpar.

Örnek: `3 * 4` sonucu 12'dir.

x / y Bölme(Divide): Bölme işlemi yapar. İlk sayıyı ikinci sayıya böler. Sonuç genellikle kayan noktalı bir sayıdır (Float).

Örnek: `10 / 4` işleminin sonucu `2.5` olur. `6 / 3` işleminin sonucu ise `2.0` olur (tam bölme durumunda bile genellikle kayan noktalı bir tipte sonuç elde edilir).

x ÷ y Tam Sayı Bölme(Integer Divide): x'in y'ye bölümünün tam sayı kısmını verir. Sonucun ondalık kısmı atılır.

Örnek: `10 ÷ 4` işleminin sonucu `2` olur (ondalık kısmı atılarak). `7 ÷ 3` işleminin sonucu da `2` olur.

Not: Bu operatörün yazımı için klavyede '\div' ifadesi yazıldıktan sonra Tab tuşuna basılması gerekmektedir; zira bu karakter klavyede varsayılan olarak bulunmamaktadır..

x \y Ters Bölme(Inverse divide): İkinci sayıyı ilk sayıya böler (y/x).

Örnek: `2\10` işleminin sonucu `0.2` iken, `10\2` işleminin sonucu `5.0` olmaktadır. '\' operatörü, genel bölme operatörü ('/') gibi ilk sayının ikinciye bölümünü değil, ikinci sayının ilk sayıya bölümünü ifade eder. Bu operatör özellikle matris işlemlerinde (lineer cebirde) çözüm bulmak için kullanılmakla birlikte, temel aritmetikte de benzer bir anlama gelmektedir..

x ^y Üs Alma(power): x sayısının y. kuvvetini hesaplar.

"Örnek: `2^3` işleminin sonucu `8`'dir. `5^2` işleminin sonucu ise `25`'tir.

x % y Mod(remainder): x'in y'ye bölümünden kalanı verir. Sonucun işareti, bölünenin (x) işaretiyle aynıdır.

Örnek: `10 % 3` işleminin sonucu `1`'dir. `7 % 4` işleminin sonucu `3`'tür. `-10 % 3` işleminin sonucu ise `-1`'dir.

3.2 Mantıksal Operatörler

Bilgisayar bilimlerinde Boolean (Mantıksal) veri türü, bir ifadenin doğruluk değerini temsil eder. Bu değer yalnızca iki durumdan birini alabilir: doğru (true) veya yanlış (false). Julia, bu Boolean değerler üzerinde işlem yapmak ve mantıksal ifadeler oluşturmak için çeşitli operatörler sunar. Bu bölümde, en sık kullanılan mantıksal operatörler incelenecektir.

İfade	**İsim**	**Tanım**
`!x`	Mantıksal DEĞİL operatörü	Mantıksal değerini tersine çevirir
`x && y`	Mantıksal VE operatörü	Tüm doğruluk değerlerinin 'true' olması durumunda 'true' sonucunu verir.
`x || y`	Mantıksal VEYA operatörü	Doğruluk değerlerinden en az birinin 'true' olması durumunda 'true' sonucunu verir.

Tablo 2: Temel mantıksal operatörler.

Mantıksal DEĞİL (Not) Operatörü (!) (Negation)

Bu operatör, tek bir Boole ifadesi üzerinde işlem yaparak ifadenin mantıksal değerini tersine çevirir. Eğer ifade 'true' ise sonuç 'false', eğer ifade 'false' ise sonuç 'true' olarak elde edilir.

Girdi	Çıktı(!Girdi)
true	false
false	true

Tablo 3: Mantıksal DEĞİL (Not) Operatörü (!) (Negation)

Kısa Devre Özelliği (Short-Circuiting): && operatörü kısa devre özelliğine sahiptir. Eğer ilk ifade false olarak değerlendirilirse, sonucun zaten false olacağı belirlendiğinden ikinci ifade hiç değerlendirilmez. Bu özellik, özellikle ikinci ifadenin değerlendirilmesinin maliyetli olduğu veya yan etkiler içerebileceği durumlarda performans ve kod güvenliği açısından önem arz etmektedir.

Mantıksal VEYA (Or) Operatörü (||)(Short-Circuiting OR): Bu operatör, iki Boole ifadesini karşılaştırır. İfadelerden en az birinin true olması durumunda sonuç true olarak elde edilir. Yalnızca her iki ifadenin de false olması durumunda sonuç false olarak elde edilir.

Girdi	Çıktı(!Girdi)
true || true	true
true || false	true
false || true	true
false || false	false

Tablo 4: Mantıksal VEYA (Or) Operatörü (||)(Short-Circuiting OR)

Kısa Devre Özelliği (Short-circuiting): || operatörü de kısa devre özelliğine sahiptir. Eğer ilk ifade true olarak değerlendirilirse, sonucun zaten true olacağı belirlendiğinden ikinci ifade hiç değerlendirilmez. Bu durum, benzer şekilde performans ve kod güvenliği avantajları sunmaktadır.

Mantıksal VE (And) Operatörü (&&)(Short-Circuiting AND): Bu operatör, iki Boole (mantıksal) ifadeyi karşılaştırmak için kullanılır. Mantıksal VE operatörünün sonucu, yalnızca her iki giriş ifadesi de `true` olduğunda `true` olarak elde edilir.

Diğer tüm durumlarda (yani giriş ifadelerinden en az biri `false` olduğunda), operatörün sonucu `false` olarak elde edilir. Başka bir deyişle, bu operatörün `true` sonuç vermesi için tüm koşulların aynı anda sağlanması gerekir.

Girdi	Çıktı(!Girdi)
true || true	true
true || false	false
false || true	false
false || false	false

Tablo 5: Mantıksal VE (And) Operatörü (&&)(Short-Circuiting AND)

Kısa Devre Özelliği (Short-circuiting): && operatörü, sol taraftaki ifade `false` ise, sağ taraftaki ifadeyi asla değerlendirmez. Çünkü mantıksal VE'nin sonucu, ilk ifade `false` ise, diğer ifade ne olursa olsun her zaman `false` olacaktır.

3.3 Bitsel Operatörler

Bitsel Operatörler Nedir?

Bitsel operatörler, bilgisayarın tam sayıları sakladığı ikili (binary) sistemdeki (0 ve 1'lerden oluşan) makine dili düzeyinde doğrudan işlem yapan operatörlerdir. Normal aritmetik operatörler ($+$, $-$, $\times$, $\div$) sayılarla işlem yaparken ve mantıksal operatörler (&&,||) Boole doğruluk değerleri `true` veya `false` ile işlem yaparken, bitsel operatörler sayıyı oluşturan her bir bit ile tek tek işlem gerçekleştirir.

Bu operatörler genellikle düşük seviyeli programlama, donanım kontrolü, bayrak yönetimi (bir durumun açık/kapalı olmasını tek bir bitle gösterme) veya performans odaklı hesaplamalar için kullanılır. Julia, bu standart bitsel operatörleri tüm temel sayı tipleri üzerinde destekler. Aşağıdaki bölümde, Julia'da mevcut olan temel bitsel operatörler detaylı olarak incelenecektir.

Bu operatörler genellikle düşük seviyeli programlama, donanım kontrolü, bayrak yönetimi (bir durumun açık/kapalı olmasını tek bir bitle gösterme) veya performans odaklı hesaplamalar için kullanılır. Julia, bu standart bitsel operatörleri tüm temel sayı tipleri üzerinde destekler. Görüntüdeki tabloda ve aşağıda yer alan operatörler şunlardır:

İfade	**İsim**	**Tanım**
`x & y`	Bitsel VE	her iki bit de 1 ise 1 olur, diğer durumlarda 0 olur
`x |y`	Bitsel VEYA	karşılıklı bitlerden en az biri 1 ise 1 olur, her ikisi de 0 ise 0 olur
$x \underline{\vee} y$	Bitsel XOR	karşılıklı bitler farklı ise 1 olur (biri 0 diğeri 1), aynı ise 0 olur
$x \mathbin{\overline{\wedge}} y$	Bitsel NAND	bit, ancak her iki bit de 1 ise 0 olur, diğer durumlarda 1 olur
$x \underline{\vee} y$	Bitsel NOR	ancak her iki bit de 0 ise 1 olur, diğer durumlarda 0 olur
`x >> y`	Mantıksal Sağa Kaydırma	bitleri y sayısı kadar sağa kaydırır
`x » y`	Aritmetik Sağa Kaydırma	bitleri y sayısı kadar sağa kaydırır ama negatif sayıların işaretini korur
`x « y`	Sola Kaydırma	bitleri y sayısı kadar sola kaydırır

Tablo 6: Bitsel operatörler.

x & y Bitsel VE(Bitwise AND): Bit düzeyinde VE (Bitwise AND) operatörü, iki ikili sayının karşılık gelen bitleri üzerinde mantıksal VE işlemini uygular. Elde edilen bit değeri, yalnızca her iki giriş bitinin de '1' olması durumunda '1' olur; diğer tüm durumlarda '0' olarak belirlenir.

Örnek (basitçe): 5 & 3 -> 5 ikili olarak 0101, 3 ikili olarak 0011'dir.

x |y Bitsel VEYA(Bitwise OR): Bit düzeyinde VEYA (Bitwise OR) operatörü, iki ikili sayının bitlerini birebir eşleştirerek bir mantıksal VEYA işlemi gerçekleştirir. Bu işlem sonucunda elde edilen bit değeri, karşılık gelen giriş bitlerinden en az birinin '1' olması durumunda '1' olur; her iki giriş bitinin de '0' olması durumunda ise '0' değeri atanır.

Örnek: 5 | 3 -> 0101 | 0011 = 0111 (7)

$x \underline{\vee} y$ **Bitsel XOR(Bitwise XOR (Exclusive OR)):** Bit düzeyinde XOR (Exclusive OR) operatörü, iki ikili sayının karşılıklı bitleri üzerinde bir dışlayıcı VEYA mantığı uygular. Elde edilen bit değeri, ancak ve ancak karşılık gelen giriş bitlerinin birbirinden farklı olması durumunda '1' olur (yani biri '0' ve diğeri '1' ise); giriş bitlerinin aynı olması durumunda ise '0' değeri alır.

Örnek: $5 \underline{\vee} 3$ -> 0101 ^0011 = 0110 (6)

$x \overline{\wedge} y$ **Bitsel NAND - Değil VE(Bitwise Nand (Not AND)):** Bit düzeyinde NAND (Not AND) operatörü, bit düzeyinde VE işleminin mantıksal değillemesidir. Bu işlemin çıktısı, yalnızca her iki giriş bitinin de '1' olması durumunda '0' olur; diğer tüm kombinasyonlarda '1' değeri üretir. Bu işlem, $\tilde{}$(x & y) ifadesiyle eşdeğerdir.

$x \underline{\vee} y$ **Bitsel NOR - Değil VEYA(Bitwise Nor (Not OR)):** Bit düzeyinde NOR (Not OR) operatörü, bit düzeyinde VEYA işleminin mantıksal değillemesidir. Bu işlemin sonucu, yalnızca her iki giriş bitinin de '0' olması durumunda '1' olur; diğer tüm durumlarda '0' değeri alır. Bu işlem, $\tilde{}$(x | y) ifadesiyle eşdeğerdir.

x »> y Mantıksal Sağa Kaydırma(Logical Shift Right): Mantıksal sağa kaydırma (Logical Shift Right) işlemi, bir sayının ikili temsilindeki bitlerin 'y' miktarı kadar sağa doğru ötelenmesini sağlar. Bu işlem sırasında sol taraftan açılan boş bit konumları her zaman '0' ile doldurulur. İşaretsiz tam sayılar veya işaretli tam sayıların pozitif değerleri için '»' operatörü ile aynı davranışı sergiler.

x » y Aritmetik Sağa Kaydırma(Arithmetic Shift Right): Aritmetik sağa kaydırma (Arithmetic Shift Right) işlemi, bir sayının ikili temsilindeki bitlerin 'y' miktarı kadar sağa doğru kaydırılmasını ifade eder. Bu kaydırma sırasında, sol taraftan açılan boş bit konumları, sayının orijinal işaret biti (en soldaki bit) ile doldurulur. Bu özellik, özellikle negatif sayıların işaretini korumak için önemlidir.

x « y Sola Kaydırma(Logical/Arithmetic Shift Left): Sola kaydırma (Logical/Arithmetic Shift Left) işlemi, bir sayının ikili temsilindeki bitlerin 'y' miktarı kadar sola doğru ötelenmesini sağlar. Bu işlemde, sağ taraftan açılan boş bit konumları daima '0' ile doldurulur. Bit düzeyinde sol kaydırma, hem mantıksal hem de aritmetik kaydırma bağlamında aynı davranışı gösterir. Bu işlem, x * (2^y) formundaki çarpma işlemlerini verimli bir şekilde gerçekleştirmek için kullanılabilir.

3.4 Bileşik Atama Operatörleri

Her ikili aritmetik ve bit düzeyinde operatör, ilgili işlemin sonucunu sol taraftaki işlenene atayan bir bileşik atama operatörüne sahiptir. Bu operatörler, ilgili ikili operatörün hemen ardından atama operatörü (=) kullanılarak oluşturulur.

Örneğin, x += 3 ifadesi, x = x + 3 ifadesi ile işlevsel olarak eşdeğerdir.

Tüm ikili aritmetik ve bit düzeyinde operatörler için mevcut bileşik atama operatörleri aşağıdaki tabloda sunulmuştur:

Operatör	Uzun Hali
`+=`	`x = x + y`
`-=`	`x = x - y`
`*=`	`x = x * y`
`/=`	`x = x / y`
`\=`	`x = x \y`
`÷=`	`x = x ÷ y`
`%=`	`x = x % y`
`^=`	`x = x ^y`
`&=`	`x = x & y`
`|=`	`x = x | y`
`⩒`	`x = x ⩒ y`
`»>=`	`x = x »> y`
`»=`	`x = x » y`
`«=`	`x = x « y`

Tablo 7: Bileşik Atama Operatörleri.

4 -Veri Tipleri Ve Yapilari

Programlamanın temelini oluşturan verilerin etkin bir şekilde işlenebilmesi ve yönetilebilmesi, onların doğru bir biçimde sınıflandırılmasına ve amaca uygun yapılar içerisinde organize edilmesine bağlıdır. Bu bölümde, Julia programlama dilinin sunduğu temel veri tipleri ile bu tipler üzerine kurulu olan ve verimli algoritma tasarımına olanak tanıyan temel veri yapıları ayrıntılı bir şekilde incelenecektir.

4.1 Veri Yapıları (Data Structures) – Genel Tanıtım

Programlama dillerindeki tip sistemleri genel olarak iki ana kategoriye ayrılır: statik ve dinamik tip sistemleri. Statik tip sistemlerinde, her program ifadesinin ve değişkenin tipi, programın icrasından önce, derleme aşamasında belirlenir ve doğrulanır. Buna karşın, dinamik tip sistemlerinde ise tiplerle ilgili çıkarımlar ve doğrulamalar, programın çalışma zamanında (run-time) işlenen gerçek değerler üzerinden yapılır; yani bir değişkenin veya ifadenin tipi, kod çalışana kadar kesin olarak bilinmeyebilir. Nesne yönelimli programlama, statik tipli dillere, derleme zamanında nesnelerin kesin tiplerini bilmeksizin kod yazılmasına olanak tanıyarak bir miktar esneklik katmıştır. Bu esneklik, özellikle polimorfizm aracılığıyla sağlanır. Polimorfizm, bir temel sınıftan türetilmiş farklı alt sınıflara ait nesnelerin, ortak bir arayüz üzerinden farklı biçimlerde kullanılabilmesine imkan tanır. Klasik dinamik tipli dillerde ise kodun tamamı doğal olarak polimorfiktir: değerlerin tipleri üzerindeki kısıtlamalar, genellikle yalnızca çalışma zamanında yapılan açık tip kontrolleriyle veya bir nesnenin belirli bir işlemi destekleyip desteklemediğinin kontrolüyle ortaya çıkar.

Julia'nın tip sistemi, dinamik bir yapıya sahip olmakla birlikte, programcının isteğe bağlı olarak değerlere spesifik tipler atfetmesine olanak tanıyarak statik tip sistemlerinin bazı avantajlarını sunar. Bu özellik, yüksek performanslı kod üretimi için önemli bir yardımcı olmasının yanı sıra, daha da kritik olarak, fonksiyon argümanlarının tiplerine dayalı yöntem seçimi (method dispatch) mekanizmasının dilin temel bir parçası olarak entegre edilmesini mümkün kılar.

Julia'da, tip belirtilmediğinde varsayılan davranış, değerlerin herhangi bir tipe sahip olabilmesine izin vermektir. Bu sayede, tipler açıkça deklare edilmeksizin birçok işlevsel Julia programı yazılabilir. Bununla birlikte, daha fazla ifade gücüne veya kesinliğe ihtiyaç duyulduğunda, önceden tipsiz (untyped) olarak yazılmış koda kademeli bir şekilde açık tip beyanları (type annotations) eklemek oldukça kolaydır. Tip beyanları eklemenin üç temel amacı bulunmaktadır: Julia'nın güçlü çoklu yöntem seçimi (multiple dispatch) mekanizmasından faydalanmak, kodun insan tarafından okunabilirliğini artırmak ve olası program hatalarını erken aşamada tespit etmek.

Julia'nın tip sistemi, teknik jargonla tanımlandığında; dinamik, nominatif ve parametrik özellikler sergiler. Genel (generic) tiplerin parametrelendirilebilir ve tipler arasındaki hiyerarşik ilişkiler, yapısal alt tipleme (structural subtyping) yoluyla dolaylı olarak belirlenmek yerine, açıkça deklare edilir (nominal subtyping). Julia'nın tip sisteminin en ayırt edici özelliklerinden biri, somut (concrete) tiplerin birbirlerinin alt tipi (subtype) olamamasıdır: tüm somut tipler nihai (final) ve yalnızca soyut (abstract) tipler üst tip olarak alınabilirler. Bu yaklaşım ilk bakışta aşırı kısıtlayıcı gibi görünse de, pratikte az sayıda dezavantajla birlikte birçok önemli fayda sağlar. Programlama pratiğinde, davranışsal miras alınmasının (behavioral inheritance), yapının miras alınmasından (structural inheritance) çok daha önemli olduğu ve her ikisinin birden miras alınmasının geleneksel nesne yönelimli dillerde ciddi karmaşıklıklara yol açtığı gözlemlenmiştir. Julia'nın tip sistemine dair belirtilmesi gereken diğer önemli üst düzey nitelikler şunlardır:

- **Değerlerin Evrenselliği:** Julia'da "nesne olan" ve "nesne olmayan" değerler arasında bir ayrım bulunmaz. Dildeki her değer, tam bağlantılı tek bir tip grafiğine ait bir tipe sahip gerçek bir nesnedir ve bu grafikteki tüm tipler eşit derecede birinci sınıf varlıklardır.
- **Derleme Zamanı Tipi Kavramının Yokluğu:** Bir değerin sahip olduğu yegane tip, programın çalışması esnasındaki gerçek tipidir. Bu, nesne yönelimli dillerde genellikle "çalışma zamanı tipi" olarak adlandırılan kavrama karşılık gelir. Julia'da ise bu kavram anlamlı değildir, zira derleme zamanı ile polimorfizmin birleşimi, derleme zamanı tipi ile çalışma zamanı tipi arasında bir ayrımı gereksiz kılar.
- **Değerlerin Tipliliği, Değişkenlerin Değil:** Tipler değerlere aittir, değişkenlere değil. Değişkenler, değerlere atıfta bulunan basit isimlerdir. Bununla birlikte, ifade kolaylığı açısından zaman zaman "değişkenin tipi" ifadesi, "değişkenin işaret ettiği değerin tipi" anlamında kullanılabilir.
- **Parametrik Tipler:** Hem soyut hem de somut tipler, başka tiplerle parametrelendirilebilir. Tipler ayrıca semboller, isbitler (yani C dilindeki temel tipler veya başka nesnelere işaretçi içermeyen yapılar gibi doğrudan bellekte saklanan sayılar ve Boolean değerleri gibi) olarak kabul edilen herhangi bir tipin değerleri ve bunların demetleri (tuple) ile de parametrelendirilebilir. Tip parametreleri, kullanılmadıkları veya kısıtlanmadıkları durumlarda ihmal edilebilir.

Julia'nın tip sistemi, son derece güçlü ve ifade yeteneği yüksek olmasına karşın; anlaşılır, sezgisel ve kullanımı kolay olacak şekilde tasarlanmıştır. Birçok Julia geliştiricisi, kodlarında açıkça tip beyanları kullanma ihtiyacı duymadan etkin bir şekilde çalışabilir. Ancak, belirli programlama görevleri ve paradigmaları, deklare edilmiş tipler aracılığıyla daha anlaşılır, daha basit, daha performanslı ve daha sağlam bir yapıya kavuşabilir.

4.2 Tip Beyanları (Type Declarations)

Tip beyanları (veya tip bildirimleri), Julia programlama dilinde bir değişkene, bir fonksiyon argümanına veya bir kompozit tipin (*struct*) alanına belirli bir tipe sahip olma zorunluluğu getiren ifadelerdir. Julia, doğası gereği dinamik bir tip sistemine sahip olmakla birlikte, bu isteğe bağlı tip beyanları sayesinde statik tip sistemlerinin sunduğu bazı avantajlardan yararlanılmasını mümkün kılar.

Kullanım Amaçları ve Temel Faydaları

Tip beyanlarının kullanılmasının başlıca gerekçeleri ve sağladığı faydalar aşağıda detaylandırılmıştır:

- **Performans Optimizasyonu:** Bir değişkenin veya fonksiyon argümanının tipi derleyici tarafından bilindiğinde, Julia'nın JIT (*Just-In-Time*) derleyicisi daha optimize ve dolayısıyla daha performanslı makine kodu üretebilir. Tip kararlılığının (*type stability*) sağlanması, özellikle performans açısından kritik olan kod bölümlerinde önemli bir husustur.
- **Hata Denetimi ve Erken Teşhis:** Bir değişkene yanlış tipte bir değer atanmaya çalışıldığında veya bir fonksiyona tanımlı olmayan bir tipte argüman geçirildiğinde, Julia çalışma zamanında (*run-time*) bir `TypeError` hatası üreterek programcıyı uyarır. Bu mekanizma, hataların geliştirme sürecinin erken aşamalarında tespit edilmesine yardımcı olur.
- **Kodun Okunabilirliği ve Anlaşılırlığı:** Kod içerisinde tiplerin açıkça belirtilmesi, ilgili kod parçasının hangi tür verilerle çalıştığını ve hangi tür verileri beklediğini netleştirir. Bu durum, kodun hem yazarı hem de diğer geliştiriciler tarafından daha kolay anlaşılmasına katkıda bulunur.
- **Yöntem Seçimi (*Method Dispatch*) Mekanizması:** Fonksiyon argümanlarına tip beyanları eklenmesi, Julia'nın aynı ada sahip bir fonksiyon için farklı argüman tiplerine göre farklı davranışlar (*yöntemler*) tanımlanmasına ve çalışma zamanında bu yöntemler arasından uygun olanın seçilmesine olanak tanır (çoklu yöntem seçimi).

Sözdizimi ve Kullanım Alanları

Tip beyanları, Julia'da `::` operatörü kullanılarak gerçekleştirilir. Bu operatörün farklı kullanım alanları ve davranışları aşağıdaki gibi incelenebilir:

1. İfadeler Üzerinde Tip Onaylaması (Assertion): `::` operatörü, bir değer hesaplayan bir ifadenin sonuna eklendiğinde "bir örneğidir" (*is an instance of*) şeklinde okunur. Bu kullanım, ifadenin solundaki değerin sağdaki tipin bir örneği olduğunu doğrular. Sağdaki tip somut (concrete) ise, soldaki değerin tam olarak o tipe sahip olması gerekir; ancak sağdaki tip soyut (abstract) ise, değerin soyut tipin bir alt tipi (*subtype*) olan somut bir tip tarafından uygulanması yeterlidir. Eğer tip onayı başarısız olursa, bir istisna (*exception*) fırlatılır; aksi takdirde, sol taraftaki değer döndürülür.

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: in typeassert, expected AbstractFloat, got a value of
    type Int64
julia> (1+2)::Int
3
```

Bu, bir tip onayının herhangi bir ifadeye yerinde eklenmesine olanak tanır.

2. Değişken Beyanı (Variable Declaration): Bir atamanın sol tarafındaki bir değişkene veya bir `local` bildirimin parçası olarak eklendiğinde, `::` operatörü farklı bir anlam kazanır: değişkenin her zaman belirtilen tipe sahip olacağını bildirir. Bu durum, C gibi statik olarak tiplendirilmiş bir dildeki tip bildirimine benzerdir. Değişkene atanan her değer, `convert` fonksiyonu kullanılarak bildirilen tipe dönüştürülür.

```
function foo()
    x::Int8 = 100
    x
end

# Fonksiyon çağrısı ve tipi kontrolü
x = foo()
println(typeof(x)) # Çıktı: Int8
```

Bu özellik, bir değişkene yapılan atamalardan birinin beklenmedik şekilde tipini değiştirmesi durumunda ortaya çıkabilecek performans sorunlarını önlemek için kullanışlıdır. Bu "bildirim" davranışı yalnızca belirli bağlamlarda meydana gelir (örneğin, `local x::Int8` veya `x::Int8 = 10` gibi atama sol tarafında) ve bildirimin yapıldığı geçerli kapsamın tamamı için geçerlidir, hatta bildirimin öncesinde bile. Julia 1.8 sürümünden itibaren, tip bildirimleri global kapsamda da kullanılabilmekte, bu da global değişkenlere erişimi tip stabil hale getirmektedir.

```
x::Int = 10
println(x) % 10

x = 3.5 % Hata verir: InexactError: Int64(3.5)

function foo(y)
    global x = 15.8    % Hata verir: InexactError: Int64(15.8)
    return x + y
end

# foo(10) \# Bu satır çağrıldığında üstteki hata fırlatılır.
```

3. Fonksiyon Argümanlarında ve Dönüş Tiplerinde Tip Beyanı: Tip bildirimleri, fonksiyon tanımlarına da eklenebilir. Bu, argümanların beklenen tiplerini ve fonksiyonun döndüreceği değerin tipini netleştirir.

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

Bu fonksiyondan bir değer döndürüldüğünde, bildirilen tipe (`Float64`) bir değişkene atama yapar gibi davranır: değer her zaman `Float64` tipine dönüştürülür.

4. Kompozit Tiplerin (*Struct*) Alanlarında Tip Beyanı: Kompozit tipler (yapılar veya *struct'lar*) tanımlanırken, her bir alanın tipi açıkça beyan edilerek, o alanın her zaman belirli bir tipte veri saklaması garanti altına alınır. Bu, veri bütünlüğü ve tip kararlılığı açısından kritik öneme sahiptir.

```
struct Point2D
    x::Float64
    y::Float64
end
```

4.3 Soyut Tipler (Abstract Types): Genel Kategoriler ve Organizasyon

Soyut tipler, Julia'nın tip sisteminde doğrudan somut örneklerin (*instance*) oluşturulamadığı özel tiplerdir. Bunun yerine, benzer özelliklere veya davranışlara sahip diğer tipler için genel kategoriler veya üst-tipler olarak işlev görürler. Bu tipler, kavramsal bir sınıflandırma sağlamak ve ilgili somut tipleri mantıksal bir düzen içinde bir araya getirmek amacıyla kullanılır.

Kullanım Amaçları ve Temel Faydaları

Soyut tiplerin temel işlevleri ve sağladığı başlıca avantajlar aşağıdaki gibidir:

- **Tip Hiyerarşisi ve Kavramsal Organizasyon:** Programdaki tipler arasında mantıksal bir düzen ve sınıflandırma oluşturulmasını sağlarlar. Örneğin, Julia'da `Number` (Sayı) soyut bir tip olarak tanımlanmıştır ve bu tipin altında `Real` (Gerçel Sayı) gibi başka soyut tipler ile `Integer` (Tam Sayı) veya `AbstractFloat` (Kayan Noktalı Sayı Soyutu) gibi somut tipler bu hiyerarşinin daha özel katmanlarında yer alır. Bu yapı, tiplerin anlaşılır gruplar halinde organize edilmesine olanak tanır.
- **Polimorfizm ve Esnek Kod Geliştirme:** Fonksiyonlar, argümanları için soyut tipler belirtilerek daha genel bir yapı kazanır. Bu sayede, aynı soyut tipin altındaki farklı somut tipler üzerinde aynı fonksiyonun çalıştırılması mümkün olur. Bu mekanizma, Julia'nın ayırt edici özelliklerinden olan çoklu yöntem seçimi (*multiple dispatch*) için temel bir zemin oluşturarak, kodun daha esnek ve yeniden kullanılabilir olmasını sağlar.

- **Ortak Davranışların Belirtilmesi (Dolaylı Arayüz):** Soyut tipler, bir grup somut tipin sergilemesi beklenen genel davranışsal özellikleri dolaylı olarak belirtebilir. Bu yaklaşım, Java gibi dillerdeki arayüz (*interface*) yapılarının getirdiği katı zorunlulukları içermese de, kavramsal bir şablon sunar.

Tanımlama Sözdizimi

Soyut tipler, `abstract type SoyutTipAdi end` sözdizimi kullanılarak tanımlanır. Bir soyut tip, başka bir soyut tipten `<:` operatörü ile daha özel bir kategori olarak türeyebilir. Örneğin: `abstract type AltSoyutTipAdi <: UstSoyutTipAdi end`. Aşağıdaki Julia kodu, soyut ve somut tip tanımlamalarına bir örnek sunmaktadır:

```
abstract type GeometricShape end
abstract type PlanarShape <: GeometricShape end

struct Circle <: PlanarShape
    radius::Float64
end
```

Bu örnekte, `GeometricShape` genel bir soyut kategori, `PlanarShape` ise ondan daha özel bir soyut kategoridir. `Circle` ise `PlanarShape` soyut kategorisi altında tanımlanmış somut bir tiptir. Hem `GeometricShape` hem de `PlanarShape` tiplerinden doğrudan bir örnek oluşturulamaz.

4.4 Julia'daki Sayı Türleri (Number Types)

Julia'nın tip sistemi, sayısal değerlerin temsilinde hem esneklik hem de yüksek performans sağlamak üzere tasarlanmıştır. Bu bağlamda, Julia çeşitli soyut ve somut sayı türlerini barındırır. Sayı türleri arasındaki hiyerarşi, genel programlama paradigmalarını desteklerken, aynı zamanda farklı hassasiyet ve aralıktaki sayısal işlemleri optimize etme olanağı sunar. Temel tam sayı (*integer*), kayan noktalı (*floating-point*) ve karmaşık (*complex*) sayı türlerinin yanı sıra, rasyonel sayılar gibi daha özel sayısal temsiller de Julia'nın çekirdek kütüphanesinin bir parçasıdır. Bu bölüm, Julia'daki başlıca sayı türlerini ve bunların tip hiyerarşisi içindeki yerlerini ele alacaktır.

4.4.1 Soyut Sayı Türleri (Abstract Number Types)

Soyut tipler, örneklenemeyen ve yalnızca tip grafiğinde düğüm görevi gören, ilgili somut tiplerin kümelerini tanımlayan yapılar olarak işlev görürler. Sayısal bağlamda soyut tipler, Julia'nın tip sisteminin belkemiğini oluşturarak kavramsal bir hiyerarşi sağlarlar. Bu hiyerarşi, Julia'nın tip sistemini yalnızca bir nesne uygulamaları koleksiyonu olmaktan çıkarıp daha kapsamlı bir yapıya dönüştürür.

Örneğin, `Int8`, `UInt8`, ..., `Float64` gibi çeşitli somut sayı tipleri mevcuttur. Bu tipler farklı gösterim boyutlarına sahip olsalar da, `Int8`, `Int16`, `Int32`, `Int64` ve `Int128` gibi tiplerin tümü işaretli tam sayı tipleri olma özelliğini paylaşır. Benzer şekilde, `UInt8`, `UInt16`, `UInt32`, `UInt64` ve `UInt128` gibi tiplerin tümü işaretsiz tam sayı tipleridir; `Float16`, `Float32` ve `Float64` ise tam sayı olmaktan ziyade kayan noktalı tipler olarak ayırt edilir. Çoğu zaman bir kod parçasının, örneğin, yalnızca argümanları bir tür tam sayı ise anlamlı olması, ancak belirli bir tam sayı türüne bağlı olmaması yaygındır. Örneğin, en büyük ortak bölen algoritması tüm tam sayı türleri için çalışır, ancak kayan noktalı sayılar için çalışmaz. Soyut tipler, bir tip hiyerarşisi oluşturarak, somut tiplerin yerleşebileceği bir bağlam sağlar. Bu, örneğin, bir algoritmayı belirli bir tam sayı tipiyle kısıtlamadan, herhangi bir tam sayı tipiyle kolayca programlama yapmanıza olanak tanır.

Soyut tipler, `abstract type` anahtar kelimesi kullanılarak bildirilir. Bir soyut tipin bildirim sözdizimleri genel olarak şu şekildedir:

```
abstract type <<ad>> end
abstract type <<ad>> <: üsttip>> end
```

`abstract type` anahtar kelimesi, adı «ad» ile verilen yeni bir soyut tip tanıtır. Bu adın isteğe bağlı olarak `<:` operatörü ve halihazırda var olan bir tip ile takip edilmesi, yeni bildirilen soyut tipin bu "üst" tipin bir alt tipi olduğunu gösterir.

Bir üst tip belirtilmediğinde, varsayılan üst tip `Any`'dir – bu, tüm nesnelerin örnekleri olduğu ve tüm tiplerin alt tipleri olduğu önceden tanımlanmış bir soyut tiptir. Tip teorisinde `Any`, tip grafiğinin zirvesinde olduğu için yaygın olarak "top" olarak adlandırılır. Julia ayrıca, tip grafiğinin en altında, `Union{}` olarak yazılan önceden tanımlanmış soyut bir "bottom" tipe sahiptir. Bu, `Any`'nin tam tersidir: hiçbir nesne `Union{}`'ın bir örneği değildir ve tüm tipler `Union{}`'ın üst tipleridir.

Julia'nın sayısal hiyerarşisini oluşturan bazı soyut tipler şunlardır:

```
abstract type Number end
abstract type Real          <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer       <: Real end
abstract type Signed        <: Integer end
abstract type Unsigned      <: Integer end
```

`Number` tipi, `Any`'nin doğrudan alt tipidir ve `Real` onun alt tipidir. Buna karşılık, `Real`'in iki alt tipi (daha fazlası olsa da burada sadece ikisi gösterilmiştir) `Integer` ve `AbstractFloat` olup, dünyayı tam sayı ve reel sayı temsillerine ayırır. Reel sayı temsilleri, kayan noktalı tipleri ve rasyonel sayılar gibi diğer tipleri de içerir. `AbstractFloat` sadece reel sayıların kayan noktalı temsillerini içerir. Tam sayılar ise `Signed` ve `Unsigned` çeşitlerine ayrılır.

`<:` operatörü genel olarak "bir alt tipidir" (*is a subtype of*) anlamına gelir ve yukarıdaki gibi bildirimlerde kullanıldığında, sağdaki tipin yeni bildirilen tipin doğrudan bir üst tipi olduğunu bildirir. Ayrıca, sol işleneninin sağ işleneninin bir alt tipi olması durumunda `true` döndüren bir alt tip operatörü olarak da ifadelerde kullanılabilir:

```
julia> Integer <: Number
true
julia> Integer <: AbstractFloat
false
```

Soyut tiplerin önemli bir kullanımı, somut tipler için varsayılan uygulamalar sağlamaktır. Basit bir örnek olarak:

```
function myplus(x,y)
    x+y
end
```

Yukarıdaki argüman bildirimleri `x::Any` ve `y::Any`'ye eşdeğerdir.

Bu fonksiyon, örneğin `myplus(2,5)` olarak çağrıldığında, dispatcher verilen argümanlara uyan en spesifik `myplus` adlı yöntemi seçer. Yukarıdakinden daha spesifik bir yöntem bulunmadığı varsayıldığında, Julia daha sonra yukarıda verilen genel fonksiyona dayanarak iki `Int` argümanı için özel olarak `myplus` adlı bir yöntem tanımlar ve derler, yani örtük olarak şunu tanımlar ve derler:

```
function myplus(x::Int,y::Int)
    x+y
end
```

ve son olarak, bu spesifik yöntemi çağırır. Böylece, soyut tipler programcıların, daha sonra birçok somut tip kombinasyonu tarafından varsayılan yöntem olarak kullanılabilecek genel fonksiyonlar yazmasına olanak tanır. Çoklu yöntem seçimi sayesinde, programcı varsayılan veya daha spesifik yöntemin kullanılıp kullanılmayacağı üzerinde tam kontrole sahiptir. Programcı, argümanları soyut tipler olan bir fonksiyona güveniyorsa performansta bir kayıp olmaz, çünkü bu fonksiyon, çağrıldığı her somut argüman tipi demeti için yeniden derlenir.

Julia'nın sayısal hiyerarşisi içindeki temel soyut tipler şunlardır:

- **Number**: Tüm sayı türleri için soyut üst tiptir.
- **Real** `<: Number`: Tüm reel sayılar için soyut üst tiptir.
- **AbstractFloat** `<: Real`: Tüm kayan noktalı sayılar için soyut üst tiptir.
- **Integer** `<: Real`: Tüm tam sayılar için soyut üst tiptir (örn. `Signed`, `Unsigned` ve `Bool` dahil).

```
julia> 42 isa Integer
true

julia> 1.0 isa Integer
false

julia> isinteger(1.0)
true
```

Signed <: `Integer`: Tüm işaretli tam sayılar için soyut üst tiptir.

Unsigned <: `Integer`: Tüm işaretsiz tam sayılar için soyut üst tiptir. Dahili işaretsiz tam sayılar onaltılık (*hexadecimal*) olarak, `0x` ön ekiyle yazdırılır ve aynı şekilde girilebilir.

```
julia> typemax(UInt8)
0xff

julia> Int(0x00d)
13

julia> unsigned(true)
0x0000000000000001
```

AbstractIrrational <: `Real`: Kesin irrasyonel bir değeri temsil eden sayı tipidir. Diğer sayısal miktarlarla aritmetik işlemlerde otomatik olarak doğru hassasiyete yuvarlanır.

AbstractIrrational <: `Real`: Kesin irrasyonel bir değeri temsil eden soyut sayı tipidir. Bu tipteki sayılar, diğer sayısal miktarlarla yapılan aritmetik işlemlerde otomatik olarak doğru hassasiyete yuvarlanır. Eğer kendi özel irrasyonel sayı tipinizi (*MyIrrational*) `AbstractIrrational`'dan türetiyorsanız, bu tipin matematiksel ve tip sistemiyle uyumlu çalışabilmesi için belirli temel fonksiyonları (*metotları*) uygulaması gerekmektedir. Bu zorunlu metotlar şunlardır:

- `==(::MyIrrational, ::MyIrrational)`: İki irrasyonel değerin eşitliğini kontrol etmek için.
- `hash(x::MyIrrational, h::UInt)`: İrrasyonel değerlerin hash değerini hesaplamak için, bu özellikle koleksiyonlarda (*collections*) kullanılmak üzere önemlidir.

- `convert(::Type{F}, x::MyIrrational) where {F <: Union{BigFloat,Float32,Float64}}`: İrrasyonel değeri kayan noktalı tiplere (`BigFloat`, `Float32`, `Float64`) dönüştürmek için.

Ayrıca, eğer oluşturduğunuz bu alt tip, bazen rasyonel bir sonuç verebilecek değerleri temsil ediyorsa (örneğin, tam sayılar için $\sqrt{n}$'i temsil eden bir karekök tipi, n tam kare olduğunda rasyonel bir sayıya dönüşebilir), o zaman ek olarak bazı metotları da uygulaması önemlidir:

- `isinteger`: Değerin bir tam sayı olup olmadığını kontrol etmek için.
- `iszero`: Değerin sıfır olup olmadığını kontrol etmek için.
- `isone`: Değerin bir olup olmadığını kontrol etmek için.
- `==` operatörünün `Real` değerleriyle kullanımı: İrrasyonel tipiniz ile reel sayı tipleri arasında eşitlik kontrolü sağlamak için.

Bu ek metotlar, `AbstractIrrational` tipleri için varsayılan olarak 'false' döndüğünden, doğru davranış için uygulanmalıdır. Ayrıca, ilgili `Rational` tipinin hash değerine eşit olacak şekilde `hash` fonksiyonunun tanımlanması da gereklidir.

4.4.2 Somut Sayı Türleri (Concrete Number Types)

Julia'nın sayısal tip hiyerarşisi, performans ve esneklik dengesini sağlamak amacıyla çeşitli somut sayı türlerini içerir. Bu somut tipler, doğrudan örneklenemeyen soyut üst tiplerden türetilmiştir ve belirli boyut, hassasiyet ve davranış özelliklerine sahiptir. Aşağıda, Julia'da yaygın olarak kullanılan başlıca somut sayı türleri detaylandırılmıştır: **Kayan Noktalı Sayılar (*Floating-Point Numbers*)**

- **Float16** `<: AbstractFloat <: Real`: IEEE 754 standardına uygun, 16-bit'lik kayan noktalı sayı tipidir. İkili formatı 1 işaret biti, 5 üs biti ve 10 kesir bitinden oluşur.
- **Float32** `<: AbstractFloat <: Real`: IEEE 754 standardına uygun, 32-bit'lik kayan noktalı sayı tipidir. İkili formatı 1 işaret biti, 8 üs biti ve 23 kesir bitinden oluşur. Bilimsel gösterim için üs, küçük 'f' harfiyle girilmelidir; örneğin, `2f3` ifadesi `2.0f0 * 10^3` veya `Float32(2_000)` ile eşdeğerdir. Dizi değişmezleri ve list comprehension'lar için eleman tipi köşeli parantezden önce belirtilebilir; örn. `Float32[1,4,9]`.

- **Float64** <: `AbstractFloat` <: `Real`: IEEE 754 standardına uygun, 64-bit'lik kayan noktalı sayı tipidir. İkili formatı 1 işaret biti, 11 üs biti ve 52 kesir bitinden oluşur. Bu tip, kayan noktalı değişmezler (`1.0 isa Float64`) ve birçok işlem (`1/2`, `2pi`, `log(2)`) için varsayılan tiptir. Bilimsel gösterim için üs, 'e' veya 'E' olarak girilebilir (`2e3 === 2.0E3 === 2.0 * 10^3`). Bu kullanım, tam sayı taşmalarını önlediği için (`2.0 * 10^19 < 0` iken `2e19 > 0`) `10^n` kullanımına göre kesinlikle tercih edilmelidir.
- **BigFloat** <: `AbstractFloat`: Keyfi hassasiyetli kayan noktalı sayı tipidir.

Tam Sayılar (*Integer Numbers*)

- **Bool** <: `Integer`: `true` ve `false` değerlerini içeren Boolean tipidir. `Bool`, bir sayı türü olarak kabul edilir: `false` sayısal olarak `0`'a ve `true` sayısal olarak `1`'e eşittir. Ayrıca, `false`, `NaN` ve `Inf` değerlerine karşı çarpımsal bir "güçlü sıfır" gibi davranır.

```
julia> [true, false] == [1, 0]
true
julia> 42.0 + true
43.0
julia> 0 .* (NaN, Inf, -Inf)
(NaN, NaN, NaN)
julia> false .* (NaN, Inf, -Inf)
(0.0, 0.0, -0.0)
```

`if` ve diğer koşullu ifadeler yalnızca `Bool` tipini kabul eder; Julia'da "doğru değeri olan" (*truthy*) değerler bulunmaz. Karşılaştırmalar tipik olarak `Bool` döndürür ve broadcasted karşılaştırmalar `Array{Bool}` yerine `BitArray` döndürebilir.

- **Int8** <: `Signed` <: `Integer`: 8-bit'lik işaretli tam sayı tipidir. `-128` ile `127` arasındaki sayıları temsil eder. Bu tür tam sayıların uyarı vermeden taşabileceğine dikkat edilmelidir (`typemax(Int8) + Int8(1) < 0`).
- **UInt8** <: `Unsigned` <: `Integer`: 8-bit'lik işaretsiz tam sayı tipidir. Onaltılık (*hexadecimal*) olarak yazdırılır; örneğin, `0x07 === 7`.
- **Int16** <: `Signed` <: `Integer`: 16-bit'lik işaretli tam sayı tipidir. `-32768` ile `32767` arasındaki sayıları temsil eder. Bu tür tam sayıların uyarı vermeden taşabileceğine dikkat edilmelidir.

- **UInt16** `<: Unsigned <: Integer`: 16-bit'lik işaretsiz tam sayı tipidir. Onaltılık olarak yazdırılır; örneğin, `0x000f === 15`.
- **Int32** `<: Signed <: Integer`: 32-bit'lik işaretli tam sayı tipidir. Bu tür tam sayıların uyarı vermeden taşabileceğine dikkat edilmelidir.
- **UInt32** `<: Unsigned <: Integer`: 32-bit'lik işaretsiz tam sayı tipidir. Onaltılık olarak yazdırılır; örneğin, `0x0000001f === 31`.
- **Int64** `<: Signed <: Integer`: 64-bit'lik işaretli tam sayı tipidir. Bu tür tam sayıların uyarı vermeden taşabileceğine dikkat edilmelidir.
- **UInt64** `<: Unsigned <: Integer`: 64-bit'lik işaretsiz tam sayı tipidir. Onaltılık olarak yazdırılır; örneğin, `0x000000000000003f === 63`.
- **Int128** `<: Signed <: Integer`: 128-bit'lik işaretli tam sayı tipidir. Bu tür tam sayıların uyarı vermeden taşabileceğine dikkat edilmelidir.
- **UInt128** `<: Unsigned <: Integer`: 128-bit'lik işaretsiz tam sayı tipidir. Onaltılık olarak yazdırılır; örneğin, `0x0000000000000000000000000000007f === 127`.
- **Int**: `Sys.WORD_SIZE`-bit işaretli tam sayı tipidir ve `Int <: Signed <: Integer <: Real` hiyerarşisine uyar. Çoğu tam sayı değişmezinin varsayılan tipidir ve `Sys.WORD_SIZE`'a bağlı olarak `Int32` veya `Int64`'ün bir diğer adıdır. `length` gibi fonksiyonlar tarafından döndürülen ve dizileri indekslemek için standart tiptir. Tam sayıların uyarı vermeden taşabileceğine dikkat edilmelidir (`typemax(Int) + 1 < 0` ve `10^19 < 0`). Taşma, `BigInt` kullanılarak önlenebilir. Çok büyük tam sayı değişmezleri (`10_000_000_000_000_000_000` `isa Int128` gibi) daha geniş bir tip kullanır. Tam sayı bölmesi `div` veya `^` iken, tam sayılar üzerinde `/` işlemi `Float64` döndürür.
- **UInt**: `Sys.WORD_SIZE`-bit işaretsiz tam sayı tipidir ve `UInt <: Unsigned <: Integer` hiyerarşisine uyar. `Int` gibi, `UInt` takma adı da belirli bir bilgisayardaki `Sys.WORD_SIZE` değerine göre `UInt32` veya `UInt64`'e işaret edebilir. Onaltılık olarak yazdırılır ve ayrıştırılır; örneğin, `UInt(15) === 0x000000000000000f`.
- **BigInt** `<: Signed`: Keyfi hassasiyetli tam sayı tipidir.

Karmaşık Sayılar (*Complex Numbers*)

- **Complex{T<:Real}** <: `Number`: Reel ve sanal kısımları `T` tipinde olan karmaşık sayı tipidir. `ComplexF16`, `ComplexF32` ve `ComplexF64` sırasıyla `Complex{Float16}`, `Complex{Float32}` ve `Complex{Float64}` için diğer adlardır.

Rasyonel Sayılar (*Rational Numbers*)

- **Rational{T<:Integer}** <: `Real`: Numeratör ve paydası `T` tipinde olan rasyonel sayı tipidir. Rasyonel sayıların taşma durumu kontrol edilir.

İrrasyonel Sayılar (*Irrational Numbers*)

- **Irrational{sym}** <: `AbstractIrrational`: `sym` sembolü ile gösterilen (`pi`, `e` ve `gamma` gibi) kesin irrasyonel bir değeri temsil eden sayı tipidir.

4.5 Julia'da Dize Değişmezleri (String Literals)

Metinsel veriler, çoğu programlama dilinde olduğu gibi Julia'da da temel bir veri türünü oluşturur. Julia, karakter dizilerini ifade etmek için çeşitli yöntemler sunar ve bu yöntemlerin başında dize değişmezleri (string literals) gelir. Dize değişmezleri, kaynak kod içerisinde belirli bir metin parçasının doğrudan nasıl temsil edildiğini ve yorumlandığını belirler. Bu bölüm, Julia'da dize değişmezlerinin temel kullanımını, farklı türlerini ve bu değişmezlerle ilişkili özel davranışları akademik bir bakış açısıyla detaylandıracaktır.

4.5.1 Julia'da Dizeler (Strings)

Dizeler (strings), sonlu karakter dizileridir. Bir karakterin tam olarak ne olduğu sorusu, dize kavramının karmaşıklığını ortaya koyar. İngilizce konuşanların aşina olduğu A, B, C gibi harfler, rakamlar ve yaygın noktalama işaretleri, ASCII standardı tarafından 0 ile 127 arasındaki tam sayı değerleriyle eşleştirilerek standartlaştırılmıştır. Ancak, İngilizce dışındaki dillerde kullanılan, aksanlı ASCII karakter varyantları, Kiril ve Yunanca gibi ilgili alfabeler, Arapça, Çince, İbranice, Hintçe, Japonca ve Korece gibi ASCII ve İngilizce ile tamamen ilgisiz alfabeler de dahil olmak üzere pek çok başka karakter bulunmaktadır. Unicode standardı, bir karakterin ne olduğu karmaşıklığını ele alır ve bu sorunu çözen kesin standart olarak kabul edilir. İhtiyaçlara bağlı olarak, bu karmaşıklıkları tamamen göz ardı edip yalnızca ASCII karakterlerin var olduğunu varsaymak veya ASCII olmayan metinlerle uğraşırken karşılaşılabilecek tüm karakterleri veya kodlamaları işleyebilen kod yazmak mümkündür.

Julia, basit ASCII metinlerle uğraşmayı kolay ve verimli hale getirirken, Unicode karakterleri işlemeyi de olabildiğince basit ve verimli kılar. Özellikle, ASCII dizilerini işlemek için C tarzı dize kodu yazılabilir ve bu kod hem performans hem de semantik açıdan beklendiği gibi çalışır. Eğer böyle bir kod ASCII olmayan metinle karşılaşırsa, sessizce bozuk sonuçlar üretmek yerine, açık bir hata mesajı ile hataya düşecektir. Bu durumda, kodu ASCII olmayan verileri işleyecek şekilde değiştirmek basittir.

Julia'nın dizeleriyle ilgili dikkat çekici bazı üst düzey özellikler şunlardır:

- Julia'da dizeler (ve dize değişmezleri) için kullanılan yerleşik somut tip **`String`**'dir. Bu tip, UTF-8 kodlaması aracılığıyla tüm Unicode karakter aralığını destekler. (Diğer Unicode kodlamalarına dönüştürme veya bu kodlamalardan dönüştürme için bir `transcode` fonksiyonu sağlanmıştır.)
- Tüm dize tipleri, soyut tip olan **`AbstractString`**'in alt tipidir. Harici paketler ek `AbstractString` alt tipleri tanımlar (örn. diğer kodlamalar için). Bir dize argümanı bekleyen bir fonksiyon tanımlarken, herhangi bir dize tipini kabul etmek için tipi `AbstractString` olarak belirtilmelidir.
- C ve Java gibi dillerin aksine, çoğu dinamik dilden farklı olarak, Julia tek bir karakteri temsil etmek için birinci sınıf bir tipe sahiptir: **`AbstractChar`**. `AbstractChar`'ın yerleşik `Char` alt tipi, herhangi bir Unicode karakteri temsil edebilen 32-bit'lik bir temel tiptir (ve UTF-8 kodlamasına dayanır).
- Java'da olduğu gibi, dizeler değişmezdir (*immutable*): Bir `AbstractString` nesnesinin değeri değiştirilemez. Farklı bir dize değeri oluşturmak için, diğer dizelerin parçalarından yeni bir dize oluşturulur.
- Kavramsal olarak, bir dize, indekslerden karakterlere giden *kısmi bir fonksiyondur*: Bazı indeks değerleri için hiçbir karakter değeri döndürülmez ve bunun yerine bir istisna fırlatılır. Bu, değişken genişlikli Unicode dize kodlamaları için hem verimli hem de basit bir şekilde uygulanamayan bir karakter indeksi yerine, kodlanmış bir temsilin bayt indeksi ile dizelere verimli bir şekilde indeksleme yapılmasına olanak tanır.

4.5.2 Julia'da Karakterler (Characters)

Julia'da tek bir karakter, `Char` değeri ile temsil edilir. Bu, özel bir değişmez gösterimine ve uygun aritmetik davranışlara sahip, Unicode kod noktasını temsil eden sayısal bir değere dönüştürülebilen 32-bit'lik bir temel veri tipidir. (Julia paketleri, diğer metin kodlamaları için işlemleri optimize etmek amacıyla `AbstractChar`'ın başka alt tiplerini tanımlayabilir). `Char` değerlerinin nasıl girildiği ve gösterildiği aşağıdaki örnekte verilmiştir (karakter değişmezlerinin çift tırnak yerine tek tırnakla sınırlandığına dikkat edilmelidir):

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
julia> typeof(c)
Char
```

Bir `Char` değerini kolayca tamsayı değerine, yani kod noktasına dönüştürülebilir:

```
julia> c = Int('x')
120
julia> typeof(c)
Int64
```

32-bit mimarilerde, `typeof(c)`'nin `Int32` olacağı unutulmamalıdır. Bir tamsayı değeri tekrar bir `Char` değerine aynı kolaylıkla dönüştürülebilir:

```
julia> Char(120)
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

Tüm tamsayı değerleri geçerli Unicode kod noktaları değildir. Ancak performans nedeniyle, `Char` dönüşümü her karakter değerinin geçerli olup olmadığını kontrol etmez. Dönüştürülen her değerin geçerli bir kod noktası olup olmadığını kontrol etmek istenirse, `isvalid` fonksiyonu kullanılabilir:

```
julia> Char(0x110000)
'\U110000': Unicode U+110000 (category In: Invalid, too high)
julia> isvalid(Char, 0x110000)
false
```

Bu yazının yazıldığı an itibarıyla geçerli Unicode kod noktaları U+0000 ila U+D7FF ve U+E000 ila U+10FFFF arasındadır. Bunların tamamına henüz anlaşılır anlamlar atanmamıştır ve uygulamalar tarafından yorumlanmayabilirler, ancak tüm bu değerler geçerli Unicode karakterler olarak kabul edilir.

Tek tırnak içinde herhangi bir Unicode karakteri, dört adede kadar onaltılık basamakla başlayan `\u` veya sekiz adede kadar onaltılık basamakla başlayan `\U` kullanılarak girilebilir (en uzun geçerli değer yalnızca altı basamak gerektirir):

```
julia> '\u0'
'\0': ASCII/Unicode U+0000 (category Cc: Other, control)
julia> '\u78'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
julia> '\u2200'
'\symbol{"2200}': Unicode U+2200 (category Sm: Symbol, math)julia> '\string\U10ffff''
   \U10ffff': Unicode U+10FFFF (category Cn: Other, not assigned)
```

Julia, hangi karakterlerin olduğu gibi basılabileceğini ve hangilerinin genel, kaçışlı `\u` veya `\U` giriş formları kullanılarak çıktı verilmesi gerektiğini sistemin yerel ve dil ayarlarına göre belirler. Bu Unicode kaçış formlarına ek olarak, C'nin geleneksel kaçışlı giriş formlarının tümü de kullanılabilir:

```
julia> Int('\0')
0
julia> Int('\t')
9
julia> Int('\n')
10
julia> Int('\e')
27
julia> Int('\x7f')
127
julia> Int('\177')
127
```

`Char` değerleri ile karşılaştırmalar ve sınırlı miktarda aritmetik işlemler yapılabilir:

```
julia> 'A' < 'a'
true
julia> 'A' <= 'a' <= 'Z'
false
julia> 'A' <= 'X' <= 'Z'
true
julia> 'x' - 'a'
23
julia> 'A' + 1
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
```

4.5.3 Julia'da Dize Temelleri (String Basics)

Julia'da dize değişmezleri, çift tırnak veya üçlü çift tırnak (tek tırnak değil) ile sınırlandırılır. Örneğin:

```
julia> str = "Hello, world.\n"
"Hello, world.\n"
julia> """Contains "quote" characters"""
"Contains \"quote\" characters"
```

Dizelerdeki uzun satırlar, yeni satırdan önce bir ters eğik çizgi (\textbackslash) eklenerek bölünebilir:

```
julia> "This is a long \
       line"
"This is a long line"
```

Bir dizeden karakter çıkarmak için indeksleme kullanılır:

```
julia> str[begin]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)
julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)
julia> str[6]
',': ASCII/Unicode U+002C (category Po: Punctuation, other)
julia> str[end]
```

```
'\n': ASCII/Unicode U+000A (category Cc: Other, control)
```

Dizeler de dahil olmak üzere birçok Julia nesnesi, tamsayılarla indekslenebilir. İlk elemanın (bir dizenin ilk karakteri) indeksi `firstindex(str)` ile, son elemanın (karakter) indeksi ise `lastindex(str)` ile döndürülür. `begin` ve `end` anahtar kelimeleri, verilen boyutta sırasıyla ilk ve son indeksler için kısaltma olarak indeksleme işlemi içinde kullanılabilir. Julia'daki çoğu indeksleme gibi, dize indeksleme de 1 tabanlıdır: `firstindex` herhangi bir `AbstractString` için her zaman 1 döndürür. Ancak, aşağıda görüleceği gibi, bir dize için `lastindex(str)` genel olarak `length(str)` ile aynı değildir, çünkü bazı Unicode karakterler birden fazla "kod birimi" kaplayabilir.

end ile aritmetik ve diğer işlemler yapılabilir, tıpkı normal bir değer gibi:

```
julia> str[end-1]
'.': ASCII/Unicode U+002E (category Po: Punctuation, other)
julia> str[end/%:\symbol{"F7}%2]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

`begin`'den küçük veya end'den büyük bir indeks kullanmak hata verir:

```
julia> str[begin-1]
ERROR: BoundsError: attempt to access 14-codeunit String at index [0]
julia> str[end+1]
ERROR: BoundsError: attempt to access 14-codeunit String at index [15]
```

Ayrıca aralık indeksleme (*range indexing*) kullanarak bir alt dize çıkarılabilir:

```
julia> str[4:9]
"lo, wo"
```

`str[k]` ve `str[k:k]` ifadelerinin aynı sonucu vermediğine dikkat edilmelidir:

```
julia> str[6]
',': ASCII/Unicode U+002C (category Po: Punctuation, other)
julia> str[6:6]
","
```

İlki tek bir `Char` tipinde karakter değeri iken, ikincisi yalnızca tek bir karakter içeren bir dize değeridir. Julia'da bunlar çok farklı şeylerdir.

Aralık indeksleme, orijinal dizenin seçilen kısmının bir kopyasını oluşturur. Alternatif olarak, `SubString` tipini kullanarak bir dizeye "görünüm" (*view*) oluşturmak mümkündür. Daha basitçe, bir kod bloğunda `@views` makrosunu kullanmak, tüm dize dilimlerini alt dizelere dönüştürür. Örneğin:

```
julia> str = "long string"
"long string"
julia> substr = SubString(str, 1, 4)
"long"
julia> typeof(substr)
SubString{String}
julia> @views typeof(str[1:4]) # @views converts slices to SubStrings
SubString{String}
```

chop, chomp veya `strip` gibi birçok standart fonksiyon bir `SubString` döndürür.

4.6 Julia'da Dize Kaçış Dizileri (String Escape Sequences)

Kaçış karakterleri (*escape characters*), belirli bir işlemi gerçekleştiren özel karakterlerdir. Örneğin, `\n` karakteri yeni bir satır başlatır. Aşağıdaki `main.jl` dosyasındaki örnek, kaçış karakterlerinin kullanımını göstermektedir:

```
println("Three\t bottles of wine")
println("He said: \"I love ice skating\"")
println("Line 1:\nLine 2:\nLine 3:")
```

Yukarıdaki örnekte yer alan kaçış karakterleri şu şekilde açıklanabilir:

- `println("Three͡bottles of wine")`: Bu satırda, `\t` kaçış karakteri bir sekme (*tab*) boşluğu ekler.
- `println("He said: Ï love ice skating¨")`: Bu satırda, çift tırnak işaretlerini (") bir dize değişmezi içine yerleştirmek için " ile kaçılır.
- `println("Line 1:\nLine2:\nLine3:")` `\n` karakteri kullanılarak üç ayrı satır oluşturulur.

Bu kodun Julia'da çalıştırılmasıyla elde edilen çıktı şu şekildedir:

```
$ julia main.jl
Three    bottles of wine
He said: "I love ice skating"
Line 1:
Line 2:
Line 3:
```

4.7 Julia'da Dize İnterpolasyonu (String Interpolation)

İnterpolasyonlu bir dize (*interpolated string*), interpolasyonlu ifadeler içerebilen bir dize değişmezidir. Özel karakter `$` , bir dize değişmezindeki bir değişkenin değerine genişletilmek üzere tanımlanmasını sağlar. İfadeler `$` sınırlayıcıları içine yerleştirilir. Aşağıdaki `main.jl` dosyasındaki örnek, dize interpolasyonunun kullanımını göstermektedir:

```
name = "John Doe"
age = 34

msg = "$name is $age years old"
println(msg)

x = 12
y = 11

println("x + y = $(x + y)")
```

Yukarıdaki örnekte iki interpolasyonlu dize bulunmaktadır:

- `msg = "$name is $age years old"`: Bu dize, `name` ve `age` adlı iki değişkenin içeriğini barındırır. Bu değişkenlerin önüne `$` karakteri eklenir. Dize genişletildikten sonra, ilgili değişkenlerin değerlerini içerir.
- `println("x + y = $(x + y)")`: İnterpolasyonlu dizeler, ifadeler de içerebilir. Bu örnekte, `x + y` ifadesinin sonucu doğrudan dize içine yerleştirilmiştir.

Bu kodun Julia'da çalıştırılmasıyla elde edilen çıktı şu şekildedir:

```
$ julia main.jl
John Doe is 34 years old
x + y = 23
```

4.8 Julia'daki Yapılar (Structs)

4.8.1 Yapıların Tanımlanması (Defining a Struct)

Julia programlama dilinde, `struct` anahtar kelimesi, belirtilen alan adları ve isteğe bağlı olarak atanmış bireysel tipler temel alınarak yeni bir Bileşik Tip (Composite Type) tanımlamak için kullanılır. Varsayılan olarak, yapılar başlatıldıktan sonra değiştirilemez bir niteliktedir; yani, açıkça `mutable struct` olarak tanımlanmadıkça değişmez (immutable) olarak kabul edilirler.

Konum bilgilerini (isim, enlem ve boylam koordinatları) içeren bir `Location` tipi oluşturulmak istendiğinde, aşağıdaki yapı tanımlaması uygulanabilir:

```
struct Location
    name::String
    lat::Float32
    lon::Float32
end
```

Bir yapının değerlerle başlatılması için, varsayılan kurucu (constructor), yapının adının bir fonksiyon gibi kullanılmasıyla sağlanır:

```
loc1 = Location("Los Angeles", 34.0522,-118.2437)
```

Yapı alanlarına erişim, pek çok programlama dilinde yaygın olarak benimsenen nokta notasyonu (`.`) aracılığıyla gerçekleştirilir:

```
loc1.name    # "Los Angeles"
loc1.lat     # 34.0522
loc1.lon     # -118.2437
```

Tanımlanan bu `Location` yapısı temel alınarak, örneğin `sites` adında bir `Location` vektörü oluşturulması ve bu vektörün dinamik olarak `Location` elemanları ile doldurulması mümkündür:

```
sites = Location[]
push!(sites, Location("Los Angeles", 34.0522,-118.2437))
push!(sites, Location("Las Vegas", 36.1699,-115.1398))
```

Benzer bir sonucun İsimlendirilmiş Demetler (Named Tuples) vasıtasıyla da elde edilebileceği ifade edilebilir. Bununla birlikte, `Location` tipi gibi yeni bir veri tipinin tanıtılması, metnin açıklığını artırma ve yapısal avantajlar sunma potansiyeli taşımaktadır.

4.8.2 Değişken Yapılar (Mutable Structs)

Bir yapının bileşenlerini başlatıldıktan sonra değiştirebilme yeteneğine sahip olmak istendiğinde, aşağıda belirtildiği gibi değişken (mutable) bir yapı tanımlaması gerekmektedir:

```
mutable struct mLocation
    name::String
    lat::Float32
    lon::Float32
end
```

Bu tanımlamanın ardından, aşağıdaki işlemler gerçekleştirilebilir:

```
loc1 = mLocation("Los Angeles", 34.0522,-118.2437)
loc1.name = "LA"
```

`Base.@kwdef` makrosu, yapılarda varsayılan değerlerin ve anahtar kelime tabanlı kurucuların (keyword-based constructors) kullanımına olanak tanıyan faydalı bir araçtır. Örneğin, aşağıdaki yapılandırma benimsenebilir:

```
Base.@kwdef mutable struct Param
    Δt :: Float64 = 0.1
    n :: Int64
    m :: Int64
end
P = Param(m=50, n=35)
```

Not: Julia 1.9 sürümünden itibaren `Base` modülü `@kwdef` makrosunu dışa aktardığından, `Base.@kwdef` yerine doğrudan `@kwdef` kullanımı yeterlidir.

4.8.3 Yapıları Parçalara Ayırma (Destructuring a Struct)

Julia 1.7 sürümünden bu yana, İsimlendirilmiş Demetlerdeki (NamedTuples) yapıya oldukça benzer bir sözdizimi ile yapılar için parçalara ayırma (destructuring) özelliği mevcuttur:

```
(; n,Δt ) = P
```

Bu işlem, `P` yapısının sol taraftaki ifadelerle eşleşen tüm alanlarını alır ve bunları bir değişkene dönüştürür. Değişkenlerin sırasının önemli olmadığı ve tüm alanların parçalara ayrılmasının zorunlu olmadığı belirtilmelidir.

Yapı alanlarına daima `P.`Δt veya `P.n` şeklinde erişim sağlanabilmekle birlikte, yapıyı parçalara ayırma, Δt ve `n` değişkenlerini otomatik olarak oluşturma imkanı sunar. Bu özellik, çok sayıda alana sahip uzun bir yapıya sahip olunduğunda ve bu alanların parametre olarak kullanılması tercih edildiğinde oldukça kullanışlıdır:

```
Base.@kwdef mutable struct Param
    Δt :: Float64 = 0.1 % Δt kullanmaya devam
end

P = Param(m=50, n=35)

print("The value of Δt is $(Δt)")
# Çıktı: The value of Δt is 0.1
```

4.9 Kapsülleme (Encapsulation) ve Veri Gizleme

4.9.1 Julia'da Kapsülleme (Encapsulation in Julia)

Julia, standart yapılandırmasında kolay bir kapsülleme mekanizması sunmayan bir programlama dilidir. Açık kaynak dünyasında gelişmiş bir dil olması sebebiyle, iş birliği ve yeniden kullanılabilirlik temel öneme sahiptir. Julia, veriye veya fonksiyonlara erişimi engelleme konusunda herhangi bir kısıtlama amacı gütmemektedir.

Kurumsal ortamlarda durum farklılık göstermektedir. Özellikle C++ geliştiricileriyle, kapsüllemenin temel bir gereklilik olarak görüldüğü konularda tartışmalar yaşanmaktadır. Bu bağlamda, konuya ilişkin daha sağlam bir görüş oluşturulması önem arz etmektedir. (Bazıları, bu tür soyut konularda bir görüşe sahip olmanın kişisel bir sorumluluk olduğunu dahi düşünmektedir.)

Kapsülleme, veri ve fonksiyonları belirli bir birim (çoğunlukla bir sınıf) içinde bir araya getirme ve bu fonksiyonlardan veya verilerden bazılarına dışarıdan erişimi engelleme yeteneğini ifade eder. Dışarıdan erişilebilen bilgilere genellikle *public* (genel/açık), gizlenen bilgilere ise *private* (özel/gizli) adı verilir.

Tom Kwong'un "Julia Handbook" adlı kitabının "Robustness" (Sağlamlık) bölümünde kapsüllemeye ayrı bir bölüm ayrılmış olup, yazar kapsülleme için aşağıdaki gerekçeleri belirtmektedir:

"En Az Ayrıcalık Prensibi (Principle of Least Privilege - POLP) temel alınarak, arayüzün istemcisine gereksiz uygulama detaylarının gizlenmesi düşünülmelidir. Ancak Julia'nın veri yapısı şeffaftır – tüm alanlar otomatik olarak açığa çıkar ve erişilebilirdir. Bu durum, uygunsuz kullanım veya mutasyonların sistemi bozma potansiyeli taşıyan bir problem teşkil etmektedir. Ek olarak, alanlara doğrudan erişim yoluyla kod, nesnenin temel uygulamasıyla daha sıkı bir şekilde kenetlenmektedir." Bu bağlamda, öğelerin korunması gereken başlıca alanlar aşağıdaki şekilde özetlenebilir:

Ayrıca, kapsüllemeyi ele alan ve daha çok beklenen Nesne Yönelimli (Object-Oriented - OO) kavramlara ve bunların Julia ile ilişkisine değinen bir blog yazısı da mevcuttur.

Bu bağlamda, öğelerin korunması gereken başlıca alanlar aşağıdaki şekilde özetlenebilir:

- Fonksiyonlar (Functions)
- Tipler (Types)
- Modüller (Modules)

4.9.2 Fonksiyonlar (Functions)

Fonksiyonların temel yapısı ve işlevi genellikle bilinmekle birlikte, derinlemesine incelendiğinde belirli özellikler ortaya çıkmaktadır. Fonksiyonlar, belirli girdileri alıp, birtakım içsel operasyonlar veya diğer fonksiyon çağrıları aracılığıyla işleyerek çıktı üretirler. Ayrıca, fonksiyonlar girdileri üzerinde mutasyon (değişiklik) da gerçekleştirebilirler. Julia'da fonksiyonlar yerel bir kapsama (local scope) sahiptir; bu durum, fonksiyon dışında tanımlanan öğelerin fonksiyonun içindeki verilere veya fonksiyonlara doğrudan erişemediği anlamına gelir.

Bu bağlamda, fonksiyonlar içsel kapsülleme yeteneği sergilemektedir. Bir fonksiyon, kendi kapsamı içinde tanımlanan veri ve fonksiyonları otomatik olarak kapsülleyen işlevsel bir birim olarak değerlendirilebilir.

Aşağıdaki örnek, iç içe bir fonksiyon ve bir değişken içeren bir fonksiyonu göstermektedir. Bu içsel öğelere dışarıdan doğrudan erişim mümkün değildir:

```
function example()
    x = 5
    function inside()
        y = 6
    end
end
```

Yukarıdaki örnekte görüldüğü gibi, dışarıdan erişim denemeleri hata ile sonuçlanmaktadır:

```
julia> example.x
ERROR: type example has no field x
julia> example.inside
ERROR: type example has no field inside
```

Dolayısıyla, Julia fonksiyonlar düzeyinde kapsülleme sunmaktadır. Ancak, bu içsel öğeleri "public" (genel) hale getirmenin doğrudan bir yolu mevcut değildir; bu da istenildiğinde dahi yeniden kullanımlarının kısıtlanmasına neden olmaktadır. Temelde, bu yapılar varsayılan olarak "private" (özel) niteliktedir. Oysa, hangi öğelerin genel, hangilerinin özel olacağının seçilebilmesi arzu edilen bir özelliktir.

Aslında, fonksiyonun sadece "public" olarak sunulması gereken öğeleri döndürmesi mümkündür. Hatta, yerel değişkenleri değiştiren yerel fonksiyonları döndürerek oldukça yaratıcı yaklaşımlar geliştirilebilir. Aşağıdaki örnek bu durumu açıklamaktadır:

```
function test()
    x::Int64 = 2
    function get_x()
        return x
    end
    function add_x(y::Int64)
        x += y
        return nothing
    end
    return (get_x = get_x, add_x = add_x)
end
```

Bu örneğin çıktıları aşağıdaki gibidir:

```
julia> t = test()
julia> t.add_x(4)
julia> t.get_x()
6
```

Bir fonksiyonun yerel bir fonksiyonu döndürmesi "closure" (kapanım) olarak adlandırılır. Yukarıdaki örnekte bu yaklaşım, gereksiz karmaşıklık getirme potansiyeli taşımakla birlikte, kapsülleme işlevini yerine getirmektedir.

Fonksiyonlar tüm iş yükünü üstlenir; veri bunlar aracılığıyla işlenir ve fonksiyonlar birbirlerini çağırırlar. İnsanlar erişimi engellemek amacıyla kapsüllemeyi tartıştıklarında, genellikle belirlenmiş bir birim (çoğunlukla bir sınıf) dışındaki fonksiyonların özel verilere erişmemesi gerektiğini ifade ederler.

Bu noktadan sonra, Bileşik Tiplerin ve özelliklerinin incelenmesine geçilmesi uygun olacaktır.

4.9.3 Bileşik Tipler (Composite Types)

Julia'da veri daima bileşik tiplerde (composite types) saklanmaktadır. Bazı görüşlere göre, bu tipler metotları olmayan sınıflar olarak değerlendirilebilir. Dolayısıyla, verilere erişimi engellemek hedeflendiğinde, tipler bu işlevi yerine getirebilir.

Yalnızca yazma erişimini engellemek isteniyorsa, bu oldukça basit bir şekilde, tipin değişmez (immutable) olarak tanımlanmasıyla sağlanır ki bu, varsayılan davranıştır.

Alanların (fields) bir alt kümesine yazma erişimini engellemek gerektiğinde, verilerin değişken (mutable) ve değişmez olmak üzere iki ayrı tipe ayrıştırılması önerilir. Daha sonra bu iki tipin bir bileşik tipi oluşturulur. Bu yaklaşım, söz konusu gereksinim için yeterli bir çözüm sunmaktadır.

Okuma erişimini de engelleme ihtiyacı biraz daha karmaşıktır.

Öncelikle, kullanıcıları bu verilerin özel (private) olduğu konusunda açıkça bilgilendirmek önemlidir. Yaygın olarak kabul gören kural, alan adının önüne alt çizgi (_) eklemektir; örneğin, `_my_private_data`. Bu durum, istenmeyen erişim riskini azaltmaktadır. Ancak bu, bir tür "centilmenler anlaşması" niteliğindedir; zira kullanıcılar hala bu verilere erişebilir, fakat bunu kendi sorumluluklarında yapmış olurlar.

Bir adım daha ileri gidilerek, özel tip için `Base.getproperty` ve `Base.setproperty!` fonksiyonlarının yeni bir metodu tanımlanabilir. Bu yaklaşım, özel alanlara erişimi daha zor hale getirecektir.

```
1 struct Example
2     public
3     also_public
4     _private
```

```
end
function Base.getproperty(example::Example, prop::Symbol)
    public_props = (:public, :also_public)
    if prop in public_props
        return getfield(example, prop)
    else
        throw(UndefVarError(prop))
    end
end
```

Yukarıdaki örneğin çıktıları aşağıdaki gibidir:

```
julia> example = Example(1, 2, 3)
julia> example._private
ERROR: UndefVarError: _private not defined
```

Eğer değişken bir tipin belirli özelliklerinin kullanıcılar tarafından değiştirilmesi istenmiyorsa, `setproperty!` fonksiyonu için de benzer bir yöntem kullanılabilir. Julia'da, kendi tipinize yönelik tüm ilgili fonksiyonları "aşırı yükleyerek" (overloading), tipinizin davranışını tamamen değiştirmek mümkündür.

Peki, kullanıcı açıkça `getfield` ve `setfield` fonksiyonlarını çağırırsa ne olur? Eğer kullanıcılar verilerinize özel yollarla erişmekte ısrarcıysa, bu noktada kapsülleme kaygısından ziyade, gerçek dünya güvenlik endişeleri üzerinde durulması daha uygun olacaktır.

Önceki bölümde olduğu gibi, kapanımlar (closures) kullanarak özelliklere erişimi daha da zorlaştırmak için yaratıcı bir seçenek mevcuttur. Bu yöntemde, bir tip, özellik olarak fonksiyonlarla başlatılır. Temelde, fonksiyonlar (tamamı özel) tiplerle (tamamı genel) birleştirilir.

```
# the private stuff
struct _Rectangle
    width::Float64
    height::Float64
end
# the public stuff
struct Rectangle
    area::Function
```

```
end
# the closure trick
function Rectangle(width::Float64, height::Float64)
    self = _Rectangle(width, height)
    area() = self.width * self.height
    return Rectangle(area)
end
# it works!
julia> rect = Rectangle(4.0, 5.0)
julia> rect.area()
20
```

Bu yaklaşımda, özel özelliklere erişim oldukça zorlaşmaktadır. Bu yöntemin her zaman en iyi performansı sağlayıp sağlamadığı kesin olmamakla birlikte, entelektüel açıdan ilgi çekici bir egzersizdir.

Bu yaklaşım kişisel olarak kullanılmamış olmakla birlikte, eğer bu yol tercih edilecekse, birden fazla kurucu tanımlarken dikkatli olunmalıdır; zira hepsi aynı şekilde davranan bir tip döndürmelidir. Aksi takdirde, kullanıcılar arasında kafa karışıklığı yaşanması beklenmektedir. Tipin davranışı, artık kurucu fonksiyonda kendisine atanan yerel fonksiyonlara bağlı hale gelmektedir. Bu durum, bir tipin davranışının tip tanımının kendisinden net bir şekilde bilinmesi gerektiği ilkesiyle çelişmektedir. Ayrıca, tip özelliklerine başka bir fonksiyonun enjekte edilmesi riskine karşı kesinlikle değişken (mutable) olarak tanımlanmamalıdır.

Not: Julia'da "internal properties" (içsel özellikler) üzerine devam eden bir tartışma bulunmaktadır. Bu tartışmanın gelecekte kesin bir çözüm sunması muhtemeldir.

4.9.4 Veriye Doğrudan Erişimi Engelleme Gerekçeleri (Reasons to Avoid Direct Data Access)

Bir geliştirici tarafından tasarlanmış, anlaşılması karmaşık iç verilere sahip özel bir tip söz konusu olduğunda, kullanıcılara daha basit bir arayüz sunmak ve bu karmaşık detayları gizlemek arzulanabilir.

`DataFrame` tipi bu duruma iyi bir örnektir. Bu tip, `df[satır, sütun]` şeklinde `getindex` fonksiyonu aracılığıyla satır ve sütun seçimini sağlayan basit bir arayüze sahiptir. Sütun seçimi ve ekleme işlemleri ise `df.yeni_sütun = [1, 2, 3]` şeklinde `getproperty` ve `setproperty!` fonksiyonları üzerinden gerçekleştirilir. Bu kullanıcı dostu arayüz, dahili mekanizmaların incelenmesine gerek bırakmamaktadır.

Bunun yanı sıra, kullanıcıların arayüzü göz ardı ederek dahili yapıya erişip istenmeyen bir hataya neden olma olasılığı da bulunmaktadır.

Bir örnek üzerinden bu durumun incelenmesi faydalı olacaktır. Julia'daki `Dict` tipi, basit bir arayüze sahiptir; anahtar-değer çiftleri yine `getindex` ile eklenir. Tip, çift sayısını dahili olarak saklar ve bu sayıya `length` fonksiyonu aracılığıyla erişim sağlanır. Ancak, temel `count` özelliğini mutasyon yoluyla değiştirerek sözlüğün (dictionary) bütünlüğünü bozmak mümkündür:

```
julia> d = Dict()
julia> d[:a] = "bla"
julia> length(d)
1
julia> d.count = 4
julia> length(d)
4
julia> [v for (k,v) in d]
4-element Vector{String}:
    "bla"
    #undef
    #undef
    #undef
```

Bu tür bir durumun pratikte nadiren gözlemlendiği belirtilmelidir. Bu durum, Julia kullanıcılarının dahili yapıların belgelenmemiş olmasından dolayı `Dict` tipinin bütünlüğünü kodlarında bozmadığını göstermektedir. Dahili kaynak kodunu okuyabilecek kadar yetenekli olup da, bu yapıyı bozacak kadar dikkatsiz bir kullanıcının ortaya çıkması beklenmemektedir.

4.9.5 Modüller ve Paketler (Modules and Packages)

Fonksiyonlar (ve tip kurucular), başkaları tarafından yeniden kullanılmak üzere modüller içerisine yerleştirilir. Bir paket, kolayca kurulabilen ve kullanılabilen, iyi tanımlanmış bir modül olarak işlev görür. (Kullanıcıların başkalarına ait ham kaynak kodlarını fonksiyonlarını yeniden kullanmak amacıyla doğrudan kendi kodlarına dahil etmeleri, tavsiye edilmeyen bir yaklaşımdır.)

Dolayısıyla, fonksiyonlara erişimi engellemek istendiğinde, modüllerin bu görevi üstlenmesi gerekmektedir. Bir modül açısından bakıldığında, fonksiyonlar temel olarak erişilen veri alanları gibi işlev görürler. Bu yaklaşım, Tom Kwong tarafından kitabında açıklanmıştır. Bu yöntemde, tamamen özel (private) bir `let` bloğu, tamamen genel (public) bir modül içine sarmalanır. `let` bloğu içinde küresel (`global`) olarak tanımlanan tüm fonksiyonlar, modül kapsamı içerisinde erişilebilir olacaktır.

```
module Example
    let x = 0, y = 2
        global function add_one()
            x += 1
            return nothing
        end
        global function print_x()
            return private_print()
        end
        function private_print()
            return println(x)
        end
    end
end
```

Yukarıdaki modülün çıktısı aşağıdaki gibidir:

```
julia> Example.print_x()
0
julia> Example.add_one()
julia> Example.print_x()
1
```

Modül içindeki yerel değişkenlere ve özel fonksiyonlara dışarıdan erişim mümkün değildir:

```
julia> Example.x
ERROR: UndefVarError: x not defined

julia> Example.private_print
ERROR: UndefVarError: private_print not defined
```

Modülün `let` bloğu içerisinde hem kalıcı tiplerin hem de fonksiyonların gizlenebileceği unutulmamalıdır. Julia'da öğeleri gizlemek için genel prensip, kapsamlara (scopes) yönelik yaratıcı yaklaşımlar geliştirmektir.

4.9.6 Fonksiyonlara Doğrudan Erişimi Sınırlandırma Gerekçeleri

Belirli fonksiyonlara kullanıcı erişiminin neden sınırlandırılması gerektiği hususu çeşitli motivasyonlara dayanmaktadır.

Bir fonksiyon, genel uygulama programlama arayüzünün (API) bir parçası olmayan ve gelecekteki sürümlerde değişiklik gösterebilecek dahili bir implementasyon detayı olabilir. Bu gibi durumlarda, fonksiyon için kullanıcı odaklı bir dokümantasyon dizesi (`docstring`) oluşturulmamış veya dışa aktarılmamış olabilir. Ek bir bildirim olarak, fonksiyon adının önüne alt çizgi (_) eklenmesi yaygın bir uygulamadır. Bu tür işaretler, iyi niyetli kullanıcılar için bu dahili fonksiyonu kullanmanın potansiyel risklerini anlamaları açısından yeterli bir uyarı niteliği taşımalıdır.

Geliştiricilerin kendi kod tabanları içerisinde karmaşık ve iç içe geçmiş fonksiyon çağrısı bağımlılıklarından kaçınma hedefi olabilir. Başka bir deyişle, kod bileşenleri arasındaki bağımlılığın (coupling) azaltılması amaçlanmaktadır. Bu durumda, ilgili fonksiyonların genel paket API'sine dahil edilmemesi önem arz eder. Zira, etkili bir arayüz tasarımı detaylı analiz ve düşünmeyi gerektirir. Fonksiyonları ve öğeleri gelişi güzel bir şekilde özel hale getirmek, genel sistemin kalitesine katkı sağlamayacaktır.

Eğer kullanıcılar, paketin dahili fonksiyonlarına belirgin bir ilgi gösteriyorsa, bu durumun geliştiricilerle müzakere edilmesi faydalı olabilir. Bu tür durumlarda, resmi API'nin genişletilmesi veya ilgili fonksiyonların ayrı bir pakete taşınarak resmi olarak yönetilmesi alternatif çözümler olarak değerlendirilebilir.

5 -Konrol Akışı

Programlama dillerinde kontrol akışı, programın yürütülme sırasını ve koşullara bağlı olarak farklı yollar izleyebilme yeteneğini ifade eder. Bu, algoritmaların belirli koşullar altında dallanmasını, belirli kod bloklarının tekrarlanmasını ve programın genel davranışını yönlendirmesini sağlayan temel bir mekanizmadır. Kontrol akışı yapıları, programların mantıksal tutarlılığını ve dinamik davranışını temin ederek karmaşık görevlerin etkin bir şekilde yerine getirilmesine olanak tanır. Bu bölümde, Julia programlama dilinde temel kontrol akışı yapıları olan döngüler, koşul ifadeleri, dizeler, diziler ve veri kapsayıcıları detaylı olarak incelenecektir.

5.1 Döngüler (Loops)

Programlama dillerinde ifadelerin tekrarlı değerlendirilmesi için iki temel yapı mevcuttur: `while` döngüsü ve `for` döngüsü.

Aşağıda bir `while` döngüsünün kullanımını gösteren Julia kodu örneği verilmiştir:

```
julia> i = 1;julia> while i <= 3
           println(i)
           global i += 1
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
1
2
3
```

`while` döngüsü, koşul ifadesini (bu örnekte `i <= 3`) değerlendirir ve bu ifade `true` olduğu sürece döngü gövdesini yürütmeye devam eder. Eğer koşul ifadesi, `while` döngüsüne ilk ulaşıldığında `false` ise, döngü gövdesi hiçbir zaman değerlendirilmez.

`for` döngüsü, sıkça karşılaşılan tekrarlı değerlendirme paradigmalarını daha kolay bir şekilde yazmayı sağlar. Yukarıdaki `while` döngüsünde olduğu gibi artan veya azalan sayma işlemleri oldukça yaygın olduğundan, bu işlem bir `for` döngüsü ile daha kısa ve öz bir biçimde ifade edilebilir:

Aşağıda, bir `for` döngüsü kullanarak sayıları yinelme örneği sunulmuştur:

```
julia> for i = 1:3
           println(i)
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
1
2
3
```

Burada `1:3` ifadesi, 1, 2, 3 sayı dizisini temsil eden bir `range` (aralık) nesnesidir. `for` döngüsü bu değerler üzerinde yinelenir ve her bir değeri sırayla `i` değişkenine atar. Genel olarak, `for` yapısı, `1:3` veya `1:3:13` (her 3. tam sayıyı, yani 1, 4, 7, ..., 13'ü gösteren bir `StepRange`) gibi bir aralık nesnesinden, diziler de dahil olmak üzere kullanıcı kodu veya harici paketler tarafından tanımlanan yineleyiciler gibi daha genel kapsayıcılara kadar herhangi bir "yinelenebilir" (iterable) nesne üzerinde döngü kurabilir. Aralıklar dışındaki kapsayıcılar için, = yerine genellikle `in` veya $x \in A$ anahtar kelimesi (tamamen eşdeğer olmakla birlikte) kullanılır, çünkü bu, kodun okunabilirliğini artırır:

Aşağıdaki örnekler, `in` ve $x \in A$ anahtar kelimelerinin yinelenebilir kapsayıcılarla kullanımını göstermektedir:

```
julia> for i in [1,4,0]
           println(i)
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
1
4
0
```

```
julia> for s ∈ ["foo","bar","baz"]
```

```
            println(s)
        end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
foo
bar
baz
```

Çeşitli yinelenebilir kapsayıcı tipleri, kılavuzun ilerleyen bölümlerinde tanıtılacak ve tartışılacaktır (bkz. örneğin, Çok Boyutlu Diziler (Multi-dimensional Arrays)).

Önceki `while` döngüsü formu ile `for` döngüsü formu arasındaki önemli bir fark, değişkenin görünür olduğu kapsamdır. Bir `for` döngüsü, kapsayan kapsamda aynı ada sahip bir değişkenin var olup olmadığına bakılmaksızın, her zaman kendi gövdesinde yeni bir yineleme değişkeni tanımlar. Bu durum, bir yandan `i` değişkeninin döngüden önce tanımlanması gerekmediği anlamına gelirken, diğer yandan döngü dışında görünür olmayacağı veya aynı ada sahip dış bir değişkenin etkilenmeyeceği anlamına gelir. Bunu test etmek için ya yeni bir etkileşimli oturum örneğine ya da farklı bir değişken adına ihtiyaç duyulacaktır:

Aşağıdaki Julia kodu, `for` döngüsü içindeki bir değişkenin kapsamını göstermektedir:

```
julia> for j = 1:3
            println(j)
        end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
1
2
3
```

Döngü dışında `j` değişkenine erişim girişimi, ‘UndefVarError‘ hatasına neden olur:

```
julia> j
ERROR: UndefVarError: ‘j‘ not defined in ‘Main‘
```

Aşağıdaki örnekte ise, dış kapsamda tanımlı bir değişkenin `for` döngüsü içerisindeki davranışına yer verilmiştir:

```
julia> j = 0;julia> for j = 1:3
           println(j)
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
1
2
```

Döngü sonrasında dış kapsamdaki ‘j‘ değişkeninin değeri:

```
julia> j
0
```

Mevcut yerel bir değişkeni yeniden kullanmak ve yukarıdaki davranışın değişmesini sağlamak için `for outer` yapısı kullanılabilir. Değişken kapsamı, `outer` anahtar kelimesi ve Julia'da nasıl çalıştığına dair detaylı bir açıklama için Değişkenlerin Kapsamı (Scope of Variables) bölümüne bakınız.

Bazen, bir `while` döngüsünün test koşulu `false` olmadan önce sonlandırılması veya bir `for` döngüsünün yinelenebilir nesnenin sonuna ulaşılmadan önce durdurulması uygun olabilir. Bu, `break` anahtar kelimesi ile gerçekleştirilebilir:

Aşağıdaki kod örnekleri, `break` anahtar kelimesinin `while` ve `for` döngülerinde erken sonlandırma için nasıl kullanıldığını göstermektedir:

```
julia> i = 1;julia> while true
           println(i)
           if i >= 3
               break
           end
           global i += 1
```

```
        end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
1
2
3
```

```
julia> for j = 1:1000
           println(j)
           if j >= 3
               break
           end
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
1
2
3
```

`break` anahtar kelimesi olmadan, yukarıdaki `while` döngüsü asla kendi başına sonlanmayacak, `for` döngüsü ise 1000'e kadar yinelenecekti. Bu döngülerin her ikisi de `break` kullanılarak erken sonlandırılmıştır.

Diğer durumlarda, bir yinelemeyi durdurup hemen bir sonrakine geçebilmek kullanışlıdır. `continue` anahtar kelimesi bu işlevi yerine getirir:

Aşağıdaki örnek, `continue` anahtar kelimesinin döngü içinde belirli koşullarda bir sonraki yinelemeye geçmek için nasıl kullanıldığını göstermektedir:

```
julia> for i = 1:10
           if i % 3 != 0
               continue
           end
           println(i)
```

```
6       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
3
6
9
```

Bu örnek, koşulu olumsuzlayıp `println` çağrısını `if` bloğunun içine yerleştirerek aynı davranışın daha açık bir şekilde üretilebileceği biraz kurgusal bir örnektir. Gerçekçi kullanımlarda, `continue` sonrasında değerlendirilecek daha fazla kod bulunur ve genellikle `continue` çağrısının yapıldığı birden fazla nokta olabilir.

Çoklu iç içe `for` döngüleri, yinelenebilir nesnelerinin kartezyen çarpımını oluşturacak şekilde tek bir dış döngüde birleştirilebilir:

Aşağıdaki Julia kodu, iç içe `for` döngülerinin tek bir ifadeyle nasıl birleştirilebileceğini ve kartezyen çarpım prensibini göstermektedir:

```
julia> for i = 1:2, j = 3:4
           println((i, j))
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

Bu sözdizimiyle, yinelenebilir nesneler hala dış döngü değişkenlerine referans verebilir; örneğin, `for i = 1:n, j = 1:i` geçerlidir. Ancak, bu tür bir döngü içindeki bir `break` ifadesi, sadece iç döngüyü değil, tüm iç içe döngüleri sonlandırır. Her iki değişken (`i` ve `j`), iç döngünün her çalıştığında mevcut yineleme değerlerine atanır. Bu nedenle, `i` değişkenine yapılan atamalar sonraki yinelemelerde görünür olmayacaktır:

Aşağıdaki örnek, iç içe döngülerde değişken atamasının sonraki yinelemelere etkisini göstermektedir:

```
julia> for i = 1:2, j = 3:4
           println((i, j))
           i = 0
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

Eğer bu örnek, her bir değişken için ayrı bir `for` anahtar kelimesi kullanılarak yeniden yazılsaydı, çıktı farklı olurdu: ikinci ve dördüncü değerler `0` içerecekti.

Birden fazla kapsayıcı, tek bir `for` döngüsünde `zip` fonksiyonu kullanılarak aynı anda yinelenebilir:

Aşağıdaki Julia kodu, `zip` fonksiyonunun birden fazla yinelenebilir nesneyi aynı anda döngüye sokmak için nasıl kullanıldığını göstermektedir:

```
julia> for (j, k) in zip([1 2 3], [4 5 6 7])
           println((j,k))
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
(1, 4)
(2, 5)
(3, 6)
```

`zip` kullanımı, kendisine geçirilen kapsayıcılar için alt-yineleyiciler içeren bir demet (tuple) olan bir yineleyici oluşturacaktır. `zip` yineleyicisi, `for` döngüsünün i. yinelemesinde her bir alt-yineleyicinin i. elemanını seçerek tüm alt-yineleyiciler üzerinde sırayla yinelenir. Alt-yineleyicilerden herhangi biri tükendiğinde, `for` döngüsü durur.

5.2 Koşul İfadeleri (Conditional Evaluation)

Koşullu değerlendirme, kodun belirli bölümlerinin bir boole ifadesinin değerine bağlı olarak yürütülmesini veya yürütülmemesini sağlar. Aşağıda `if-elseif-else` koşullu sözdiziminin yapısı gösterilmiştir:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

Eğer `x < y` koşul ifadesi `true` ise, ilgili kod bloğu değerlendirilir; aksi takdirde `x > y` koşul ifadesi değerlendirilir ve eğer `true` ise, ilgili blok yürütülür; bu ifadelerden hiçbiri `true` değilse, `else` bloğu değerlendirilir. Aşağıda bu yapının pratik uygulaması gösterilmiştir:

```
julia> function test(x, y)
           if x < y
               println("x is less than y")
           elseif x > y
               println("x is greater than y")
           else
               println("x is equal to y")
           end
       end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
test (generic function with 1 method)
```

Fonksiyonun farklı girdilerle test edilmesi:

```
julia> test(1, 2)
```

Çıktı:

```
x is less than y
```

```
julia> test(2, 1)
```

Çıktı:

```
x is greater than y
```

```
julia> test(1, 1)
```

Çıktı:

```
x is equal to y
```

`elseif` ve `else` blokları isteğe bağlıdır ve istenildiği kadar `elseif` bloğu kullanılabilir. `if-elseif-else` yapısındaki koşul ifadeleri, ilki `true` olarak değerlendirilene kadar sırasıyla değerlendirilir; bunun ardından ilgili blok yürütülür ve başka hiçbir koşul ifadesi veya blok değerlendirilmez.

`if` blokları "sızdırıcı" (*leaky*) bir özelliğe sahiptir; yani yeni bir yerel kapsam (local scope) oluşturmazlar. Bu, `if` koşulları içinde tanımlanan yeni değişkenlerin, daha önce tanımlanmamış olsalar bile, `if` bloğundan sonra da kullanılabileceği anlamına gelir. Bu nedenle, yukarıdaki `test` fonksiyonu aşağıdaki gibi de tanımlanabilirdi:

Aşağıdaki Julia kodu, `if` bloğu içinde tanımlanan bir değişkenin dış kapsamda nasıl kullanılabileceğini göstermektedir:

```
julia> function test(x,y)
           if x < y
               relation = "less than"
```

```
        elseif x == y
            relation = "equal to"
        else
            relation = "greater than"
        end
        println("x is ", relation, " y.")
    end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
test (generic function with 1 method)
```

Fonksiyonun çalıştırılması:

```
julia> test(2, 1)
```

Çıktı:

```
x is greater than y.
```

`relation` değişkeni `if` bloğu içinde bildirilmiş, ancak dışında kullanılmıştır. Ancak, bu davranışa dayanırken, olası tüm kod yollarının değişken için bir değer tanımladığından emin olunmalıdır. Yukarıdaki fonksiyonda yapılacak aşağıdaki değişiklik, bir çalışma zamanı hatasına neden olur:

Aşağıdaki Julia kodu, tüm kod yollarının bir değişkeni tanımlamaması durumunda ortaya çıkabilecek bir çalışma zamanı hatasını örneklendirmektedir:

```
julia> function test(x,y)
        if x < y
            relation = "less than"
        elseif x == y
            relation = "equal to"
        end
        println("x is ", relation, " y.")
    end
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
test (generic function with 1 method)
```

Fonksiyonun çalıştırılması:

```
julia> test(1,2)
```

Çıktı:

```
x is less than y.
```

Fonksiyonun bir hata ile sonuçlanan çağrısı:

```
julia> test(2,1)
```

Çıktı:

```
ERROR: UndefVarError: ‘relation‘ not defined in local scope
Stacktrace:
 [1] test(::Int64, ::Int64) at ./none:7
```

`if` blokları aynı zamanda bir değer döndürür, bu durum diğer birçok dilden gelen kullanıcılar için sezgisel olmayabilir. Bu değer, seçilen dalda en son yürütülen ifadenin döndürdüğü değerdir, örneğin:

Aşağıdaki Julia kodu, bir `if` ifadesinin nasıl bir değer döndürebileceğini göstermektedir:

```
julia> x = 33
```

Çıktı:

```
33
```

```
julia> if x > 0
           "positive!"
       else
```

```
        "negative..."
    end
```

Çıktı:

```
"positive!"
```

Çok kısa koşullu ifadelerin (tek satırlıklar), bir sonraki bölümde ana hatları belirtilen Kısa Devre Değerlendirmesi (Short-Circuit Evaluation) kullanılarak sıkça ifade edildiği unutulmamalıdır.

C, MATLAB, Perl, Python ve Ruby'den farklı olarak – ancak Java ve bazı diğer katı, tipli diller gibi – koşullu bir ifadenin değeri `true` veya `false` dışında herhangi bir şey olursa hata verir:

Aşağıdaki Julia kodu, koşullu ifadelerde boole olmayan bir değer kullanıldığında ortaya çıkan tip hatasını göstermektedir:

```
julia> if 1
           println("true")
       end
```

Çıktı:

```
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

Bu hata, koşullu ifadenin yanlış tipte olduğunu gösterir: gerekli olan `Bool` yerine `Int64` kullanılmıştır.

Üç operatör alması nedeniyle birçok dildeki tek operatör olması sebebiyle "üçlü operatör" (*ternary operator*) olarak adlandırılan `?:` yapısı, `if-elseif-else` sözdizimi ile yakından ilişkilidir, ancak daha uzun kod bloklarının koşullu olarak yürütülmesinin aksine, tek ifade değerleri arasında koşullu bir seçim gerektiğinde kullanılır. Yapısı şu şekildedir:

```
a ? b : c
```

? işaretinden önceki a ifadesi bir koşul ifadesidir ve üçlü işlem, : işaretinden önceki b ifadesini, a koşulu true ise; veya : işaretinden sonraki c ifadesini, koşul false ise değerlendirir. ? ve : etrafındaki boşlukların zorunlu olduğu unutulmamalıdır: a?b:c gibi bir ifade geçerli bir üçlü ifade değildir (ancak hem ? hem de : sonrası yeni bir satır kabul edilebilir).

Bu davranışı anlamanın en kolay yolu bir örnek görmektir. Önceki örnekte, println çağrısı her üç dal tarafından da paylaşılmaktadır: tek gerçek seçim hangi değişmez dizenin (*literal string*) yazdırılacağıdır. Bu, üçlü operatör kullanılarak daha kısa bir şekilde yazılabilir. Açıklık sağlamak adına, önce iki yollu bir versiyonu deneyelim:

Aşağıdaki Julia kodu, üçlü operatörün iki seçenekli bir koşulu nasıl daha kısa bir şekilde ifade edebileceğini göstermektedir:

```
julia> x = 1; y = 2;julia> println(x < y ? "less than" : "not less than")
```

Çıktı:

```
less than
```

```
julia> x = 1; y = 0;julia> println(x < y ? "less than" : "not less than")
```

Çıktı:

```
not less than
```

Eğer x < y ifadesi true ise, tüm üçlü operatör ifadesi "less than" dizesine değerlendirilir, aksi takdirde "not less than" dizesine değerlendirilir. Orijinal üç yollu örnek, üçlü operatörün birden fazla kullanımını zincirlemeyi gerektirir:

Aşağıdaki Julia kodu, üçlü operatörün birden fazla koşul için nasıl zincirlenebileceğini göstermektedir:

```
julia> test(x, y) = println(x < y ? "x is less than y"    :
                            x > y ? "x is greater than y" : "x is equal to
                                y")
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
test (generic function with 1 method)
```

Fonksiyonun farklı girdilerle test edilmesi:

```
julia> test(1, 2)
```

Çıktı:

```
x is less than y
```

```
julia> test(2, 1)
```

Çıktı:

```
x is greater than y
```

```
julia> test(1, 1)
```

Çıktı:

```
x is equal to y
```

Zincirlemeyi kolaylaştırmak için, operatör sağdan sola doğru ilişkilendirilir (associates).

`if-elseif-else` yapısında olduğu gibi, : öncesi ve sonrası ifadelerin yalnızca koşul ifadesi sırasıyla `true` veya `false` olarak değerlendirilirse yürütüldüğü önemlidir:

Aşağıdaki Julia kodu, üçlü operatörün "kısa devre" davranışını göstererek, yalnızca ilgili dalın yürütüldüğünü ortaya koymaktadır:

```
julia> v(x) = (println(x); x)
```

Yukarıdaki kodun çıktısı şu şekildedir:

```
v (generic function with 1 method)
```

```
julia> 1 < 2 ? v("yes") : v("no")
```

Çıktı:

```
yes
"yes"
```

```
julia> 1 > 2 ? v("yes") : v("no")
```

Çıktı:

```
no
"no"
```

6 -Koleksiyonlar ve Veri Yapıları (Collections and Data Structures)

Programlama dillerinde verilerin düzenlenmesi ve işlenmesi için çeşitli koleksiyon ve veri yapıları kullanılır. Julia'da dizeler (strings), diziler (arrays) ve diğer veri kapsayıcıları bu amaçla büyük önem taşır.

6.1 İterasyon (Iteration)

Sıralı yineleme (*sequential iteration*), `iterate` fonksiyonu aracılığıyla gerçekleştirilir. Genel `for` döngüsü, aşağıdaki yapıya sahiptir:

```
for i in iter   # or  "for i = iter"
    # body
end
```

Bu döngü, aşağıdaki yapıya çevrilir:

```
next = iterate(iter)
while next !== nothing
    (i, state) = next
    # body
    next = iterate(iter, state)
end
```

`state` nesnesi herhangi bir şey olabilir ve her yinelenebilir tip (*iterable type*) için uygun şekilde seçilmelidir. Özel bir yinelenebilir tipin tanımlanması hakkında daha fazla detay için iterasyon arayüzü kılavuzuna bakınız.

6.1.1 Base.iterate Fonksiyonu

`Base.iterate` fonksiyonu, bir yineleyiciyi bir sonraki elemanı elde etmek için ilerletir.

```
iterate(iter [, state]) -> Union{Nothing, Tuple{Any, Any}}
```

Eğer geriye kalan eleman yoksa, `nothing` döndürülmelidir. Aksi takdirde, bir sonraki eleman ve yeni yineleme durumu olmak üzere 2-elemanlı bir demet (*tuple*) döndürülmelidir.

6.1.2 Base.IteratorSize Tipi

`Base.IteratorSize` tipi, bir yineleyicinin tipine göre aşağıdaki değerlerden birini döndürür:

- `SizeUnknown()`: Uzunluk (eleman sayısı) önceden belirlenemezse.
- `HasLength()`: Sabit, sonlu bir uzunluk varsa.
- `HasShapeN()`: Bilinen bir uzunluk ve çok boyutlu bir şekil kavramı varsa (bir dizi gibi). Bu durumda `N` boyut sayısını vermelidir ve `axes` fonksiyonu yineleyici için geçerlidir.
- `IsInfinite()`: Yineleyici sonsuza kadar değerler üretiyorsa.

Bu fonksiyonu tanımlamayan yineleyiciler için varsayılan değer `HasLength()` şeklindedir. Bu, çoğu yineleyicinin `length` fonksiyonunu uyguladığı varsayılır anlamına gelir.

Bu özellik (trait), genellikle sonuç için alanı önceden ayıran algoritmalar ile sonuçlarını artımlı olarak yeniden boyutlandıran algoritmalar arasında seçim yapmak için kullanılır.

Aşağıdaki Julia kodu, `Base.IteratorSize` fonksiyonunun kullanım örneklerini göstermektedir:

```
julia> Base.IteratorSize(1:5)
```

Çıktı:

```
Base.HasShape{1}()
```

```
julia> Base.IteratorSize((2,3))
```

Çıktı:

```
Base.HasLength()
```

6.1.3 Base.IteratorEltype Tipi

`Base.IteratorEltype` tipi, bir yineleyicinin tipine göre aşağıdaki değerlerden birini döndürür:

- `EltypeUnknown()`: Yineleyici tarafından üretilen elemanların tipi önceden bilinmiyorsa.
- `HasEltype()`: Eleman tipi biliniyorsa ve `eltype` anlamlı bir değer döndürüyorsa.

`HasEltype()` varsayılan değerdir, çünkü yineleyicilerin `eltype` fonksiyonunu uyguladığı varsayılır.

Bu özellik (trait), genellikle belirli bir sonuç tipini önceden ayıran algoritmalar ile üretilen değerlerin tiplerine göre bir sonuç tipi seçen algoritmalar arasında seçim yapmak için kullanılır.

Aşağıdaki Julia kodu, `Base.IteratorEltype` fonksiyonunun kullanım örneğini göstermektedir:

```
julia> Base.IteratorEltype(1:5)
```

Çıktı:

```
Base.HasEltype()
```

Tam Olarak Uygulananlar:

Aşağıdaki yapılar `iterate` arayüzünü tam olarak uygulamaktadır:

- `AbstractRange`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`

- `BitSet`
- `IdDict`
- `Dict`
- `WeakKeyDict`
- `EachLine`
- `AbstractString`
- `Set`
- `Pair`
- `NamedTuple`

6.2 Yapıcılar ve Tipler (Constructors and Types)

Julia'da çeşitli veri yapıları, belirli kurallara ve özelliklere sahip tipler aracılığıyla oluşturulur. Bu bölümde, özellikle aralık (range) tiplerinin nasıl tanımlandığı ve kullanıldığı incelenecektir.

6.2.1 Base.AbstractRange Tipi

`AbstractRangeT <: AbstractVectorT`. Bu, `T` tipinde elemanlara sahip doğrusal aralıklar için bir üst tiptir. `UnitRange`, `LinRange` ve diğer tipler bunun alt tipleridir. Tüm alt tipler `step` değerini tanımlamalıdır. Bu nedenle `LogRange`, `AbstractRange`'in bir alt tipi değildir.

6.2.2 Base.OrdinalRange Tipi

`OrdinalRangeT, S <: AbstractRangeT`. Bu, `T` tipinde elemanlara ve `S` tipinde adım(lar)a sahip sıralı aralıklar için bir üst tiptir. Adımlar her zaman `oneunit`'in tam katları olmalı ve `T` "ayrık" bir tip olmalıdır, yani `oneunit(T)`'den daha küçük değerlere sahip olamaz. Örneğin, `Integer` veya `Date` tipleri uygunken, `Float64` uygun değildir (çünkü bu tip `oneunit(Float64)`'ten daha küçük değerleri temsil edebilir). `UnitRange`, `StepRange` ve diğer tipler bunun alt tipleridir.

6.2.3 Base.AbstractUnitRange Tipi

`AbstractUnitRangeT <: OrdinalRangeT, T`. Bu, `T` tipinde elemanlara ve `oneunit(T)` adım boyutuna sahip aralıklar için bir üst tiptir. `UnitRange` ve diğer tipler bunun alt tipleridir.

6.2.4 Base.StepRange Tipi

`StepRangeT, S <: OrdinalRangeT, S`. Bu, `T` tipinde elemanlara ve `S` tipinde aralığa sahip aralıkları temsil eder. Her eleman arasındaki adım sabittir ve aralık, `T` tipinde bir başlangıç (`start`) ve duruş (`stop`) noktası ile `S` tipinde bir adım (`step`) ile tanımlanır. Ne `T` ne de `S` kayan noktalı tipler olmamalıdır. `a:b:c` sözdizimi, `b != 0` ve `a`, `b`, `c` hepsi tam sayı olduğunda bir `StepRange` oluşturur.

Aşağıdaki Julia kodu, `StepRange` yapısının kullanım örneklerini göstermektedir:

```
julia> collect(StepRange(1, Int8(2), 10))
```

Çıktı:

```
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

```
julia> typeof(StepRange(1, Int8(2), 10))
```

Çıktı:

```
StepRange{Int64, Int8}
```

```
julia> typeof(1:3:6)
```

Çıktı:

```
StepRange{Int64, Int64}
```

6.2.5 Base.UnitRange Tipi

`UnitRangeT<:Real`. Bu, `T` tipinde bir başlangıç (`start`) ve duruş (`stop`) noktasıyla parametrelenmiş, `start`'tan `stop` aşılana kadar `1` aralıklarla elemanlarla doldurulmuş bir aralıktır. `a:b` sözdizimi, hem `a` hem de `b` tam sayı olduğunda bir `UnitRange` oluşturur.

Aşağıdaki Julia kodu, `UnitRange` yapısının kullanım örneklerini göstermektedir:

```
julia> collect(UnitRange(2.3, 5.2))
```

Çıktı:

```
3-element Vector{Float64}:
 2.3
 3.3
 4.3
```

```
julia> typeof(1:10)
```

Çıktı:

```
UnitRange{Int64}
```

6.2.6 Base.LinRange Tipi

`LinRangeT,L`. Bu, başlangıç (`start`) ve duruş (`stop`) noktaları arasında `len` kadar doğrusal olarak aralıklı eleman içeren bir aralıktır. Aralığın boyutu, bir `Integer` olması gereken `len` tarafından kontrol edilir.

Aşağıdaki Julia kodu, `LinRange` yapısının kullanım örneklerini göstermektedir:

```
julia> LinRange(1.5, 5.5, 9)
```

Çıktı:

```
9-element LinRange{Float64, Int64}:
 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5
```

`range` kullanmaya kıyasla, doğrudan bir `LinRange` oluşturmak daha az ek yüke sahip olabilir ancak kayan nokta hatalarını düzeltmeye çalışmaz.

```
julia> collect(range(-0.1, 0.3, length=5))
```

Çıktı:

```
5-element Vector{Float64}:
 -0.1
  0.0
  0.1
  0.2
  0.3
```

```
julia> collect(LinRange(-0.1, 0.3, 5))
```

Çıktı:

```
5-element Vector{Float64}:
 -0.1
 -1.3877787807814457e-17
  0.09999999999999999
  0.19999999999999998
  0.3
```

6.3 Genel Koleksiyonlar (General Collections)

Bu bölümde, Julia'daki farklı koleksiyon tipleri üzerinde uygulanabilen temel fonksiyonlar ve bunların davranışları incelenmektedir.

6.3.1 Base.isempty Fonksiyonu

`isempty(collection) -> Bool`. Bir koleksiyonun boş olup olmadığını (hiç elemanı olup olmadığını) belirler.

Uyarı: `isempty(itr)` fonksiyonu, eğer uygun bir `Base.isdone(itr)` metodu tanımlanmamışsa, durum bilgisini tutan (*stateful*) bir yineleyici `itr`'nin bir sonraki elemanını tüketebilir. Durum bilgisini tutan yineleyiciler `isdone` metodunu uygulamalıdır, ancak herhangi bir yineleyici tipini desteklemesi gereken genel kod yazarken `isempty` kullanmaktan kaçınmak isteyebilirsiniz.

Aşağıdaki Julia kodu, `isempty` fonksiyonunun kullanım örneklerini göstermektedir:

```
julia> isempty([])
```

Çıktı:

```
true
```

```
julia> isempty([1 2 3])
```

Çıktı:

```
false
```

`isempty(condition)`. Koşulda bekleyen görev yoksa `true`, aksi takdirde `false` döndürür.

6.3.2 Base.isdone Fonksiyonu

`isdone(itr, [state]) -> UnionBool, Missing`. Bu fonksiyon, yineleyici tamamlanması için hızlı bir yol ipucu sağlar. Bu, durum bilgisini tutan yineleyiciler için faydalıdır, çünkü elemanların kullanıcıya gösterilmeyecekse (örneğin `isempty` veya `zip` içinde tamamlanma durumunu kontrol ederken) tüketilmesini önlemek isterler.

Bu özelliği kullanmak isteyen durum bilgisini tutan yineleyiciler, yineleyicinin tamamlanıp tamamlanmadığına bağlı olarak `true/false` döndüren bir `isdone` metodu tanımlamalıdır. Durum bilgisini tutmayan yineleyicilerin bu fonksiyonu uygulamasına gerek yoktur.

Eğer sonuç `missing` ise, çağrıcılar kesin bir cevap elde etmek için `iterate(x, state) === nothing` ifadesini hesaplayabilirler. Ayrıca `iterate`, `isempty` fonksiyonlarına da bakınız.

6.3.3 Base.empty! Fonksiyonu

`empty!(collection) -> collection`. Bir `collection`'dan tüm elemanları kaldırır.

Aşağıdaki Julia kodu, `empty!` fonksiyonunun bir sözlüğü boşaltma örneğini göstermektedir:

```
julia> A = Dict("a" => 1, "b" => 2)
```

Çıktı:

```
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1
```

```
julia> empty!(A);
```

```
julia> A
```

Çıktı:

```
Dict{String, Int64}()
```

`empty!(c::Channel)`. Bir `Channel` c'yi dahili arabellek üzerinde `empty!` çağırarak boşaltır. Boş kanalı döndürür.

6.3.4 Base.length Fonskiyonu

`length(collection) -> Integer`. Koleksiyondaki eleman sayısını döndürür. Dizinlenebilir bir koleksiyonun son geçerli dizinini almak için `lastindex` kullanınız. Ayrıca `size`, `ndims`, `eachindex` fonksiyonlarına da bakınız.

Aşağıdaki Julia kodu, `length` fonksiyonunun farklı koleksiyonlar üzerindeki kullanım örneklerini göstermektedir:

```
julia> length(1:5)
```

Çıktı:

```
5
```

```
julia> length([1, 2, 3, 4])
```

Çıktı:

```
4
```

```
julia> length([1 2; 3 4])
```

Çıktı:

```
4
```

6.3.5 Base.checked_length Fonksiyonu

`Base.checked_length(r)`. `length(r)`'yi hesaplar, ancak sonuç `Union{Integer{etype(r)),Int}}`'e sığmadığında geçerli olabilecek taşma hatalarını kontrol edebilir.

6.3.6 Tam Olarak Uygulananlar:

Aşağıdaki yapılar `length` arayüzünü tam olarak uygulamaktadır:

- `AbstractRange`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`
- `BitSet`
- `IdDict`
- `Dict`
- `WeakKeyDict`
- `AbstractString`
- `Set`
- `NamedTuple`

7 -Çok Biçimlilik (Polymorphism)

Polimorfizm, farklı tiplerdeki varlıklara tek bir arayüz sağlanması veya birden fazla farklı türü temsil etmek üzere tek bir sembolün kullanılması prensibine dayanmaktadır. Bu kavram, temel olarak, benzer bağlamlarda farklı tiplerin işlev görmesine olanak tanıyan bir kod tekrar kullanım mekanizması sunmaktadır.

Julia gibi dinamik olarak tip tanımlayan programlama dillerinde, tip bildirilmeden tanımlanan metotlar varsayılan olarak polimorfik bir yapıya sahiptir. Bu durum, söz konusu metotların herhangi bir tipteki argümanı kabul etmesine olanak tanır; ancak, ilgili arayüzlerin implemente edilmediği bir tipin kullanılması halinde çalışma zamanında (runtime) hata oluşumu gözlemlenebilir.

Akademik kullanımda, Julia dilinin idiomatik yaklaşımı, fonksiyonları mümkün olduğunca dinamik olarak tanımlamayı ve çoklu gönderimden (multiple dispatch) faydalanmak amacıyla fonksiyonlar üzerinde tip bildirimleri kullanmayı önermektedir. Bununla birlikte, geliştirme sürecinde tip hatalarının daha erken tespit edilmesi, bildirimlerin kullanımıyla mümkündür. Bu tür hatalar yalnızca çalışma zamanında yakalanacak olsa da, programlama hatasının meydana geldiği konuma potansiyel olarak daha yakın bir noktada belirlenmelerini sağlamaktadır.

Julia'da tip bildirimlerinin kullanılması, polimorfizmden feragat edildiği anlamına gelmemektedir. Julia, çoklu gönderim aracılığıyla ad-hoc polimorfizmi, parametrik polimorfizmi (genel tip tanımlamaları ve birinci sınıf tip kurucuları ile) ve alt tiplemeyi (subtyping) (soyut tipler aracılığıyla) desteklemektedir. Bu özelliklerin tamamı, hem tip güvenliği sağlanmış hem de yüksek derecede polimorfik fonksiyonlar ve yapılar (structs) oluşturmak üzere entegre bir şekilde kullanılabilmektedir. Soyut Tiplerle Kalıtım (Inheritance with Abstract Types) Klasik nesne yönelimli programlama dillerine veya JavaScript gibi prototip tabanlı dillere aşina olanlar için, Julia'nın da kalıtım mekanizmalarına sahip olduğu belirtilmelidir. Ancak, Julia'daki kalıtım yaklaşımı, klasik nesne yönelimli dillerdeki geleneksel yapıdan farklılık göstermektedir.

Bu bağlamda, daha önceki dikdörtgen örneğimize ek olarak bir daire de dahil edilerek incelenecektir.

7.0.1 Soyut Tiplerle Kalıtım (Inheritance with Abstract Types)

Klasik nesne yönelimli programlama dillerine veya JavaScript gibi prototip tabanlı dillere aşina olanlar için, Julia'nın da kalıtım mekanizmalarına sahip olduğu belirtilmelidir. Ancak, Julia'daki kalıtım yaklaşımı, klasik nesne yönelimli dillerdeki geleneksel yapıdan farklılık göstermektedir.

Bu bağlamda, daha önceki dikdörtgen örneğimize ek olarak bir daire de dahil edilerek incelenecektir.

```
"""‘Shape‘ soyut tipinden türeyen tiplerin bir ‘alan‘ (area) metodu sağ
    laması gerekmektedir."""
abstract type Shape end
combined_area(a::Shape, b::Shape) = area(a) + area(b)

struct Circle <: Shape
    diameter::Float64
end
radius(c::Circle) = c.diameter / 2
area(c::Circle) = π * radius(c) ^ 2

"""‘AbstractRectangle‘ soyut tipinden türeyen tiplerin ‘yükseklik‘ (height)
    ve ‘genişlik‘ (width) metotlarını sağlaması beklenmektedir."""
abstract type AbstractRectangle <: Shape end
area(r::AbstractRectangle) = width(r) * height(r)

struct Rectangle <: AbstractRectangle
    width::Float64
    height::Float64
end
width(r::Rectangle) = r.width
height(r::Rectangle) = r.height

struct Square <: AbstractRectangle
    length::Float64
end
width(s::Square) = s.length
height(s::Square) = s.length
```

height (genel fonksiyon, 2 metotlu) Bu bölümde, iki soyut tipin bildirimi gerçekleştirilmiştir: Hiyerarşinin üst seviyesinde konumlanan (Any tipinin bir alt tipi olan) **Shape** ve Shape tipinin bir alt tipi olarak tanımlanan **AbstractRectangle**.

Julia'daki soyut tiplerin herhangi bir veri düzenine veya somut örneğe sahip olmadığı vurgulanmalıdır. Bu tipler, tamamen fonksiyon gönderimlerinin (function dispatches) kalıtımını sağlamak amacıyla var olurlar. Shape tipinin alt tipleri, kendi area metotlarını implemente etmeleri koşuluyla **combined_area** metodunu kullanabilmektedir. Benzer şekilde, AbstractRectangle tipinin alt tipleri, kendi height ve width metotlarını sağlamaları durumunda area metodundan faydalanabilmektedir.

Circle tipi doğrudan Shape'ten türemekte ve çapından türetilen bir area metodunu implemente etmektedir.

Rectangle ve Square tiplerinin her ikisi de AbstractRectangle'ın alt tipleri olarak tanımlanmıştır ve bu soyut tipin area metodunu kalıtırken, kendi width ve height metotlarını implemente etmektedirler. Julia'da alt tiplerde belirli arayüzlerin implementasyonunu zorlayacak bir mekanizmanın bulunmaması, bu durumun detaylı belgelendirme ihtiyacını ortaya çıkarmaktadır. Bu yaklaşımın potansiyel faydalarından biri, kullanılmayacak bir üst tip metoduna ilişkin arayüzlerin implementasyon gereksinimini ortadan kaldırmasıdır. Ancak, zorunlu bir sözleşmenin varlığı, eksik metotlar nedeniyle ortaya çıkabilecek çalışma zamanı hatalarını engellemek adına daha güvenli bir yöntem olarak değerlendirilebilir.

```
c = Circle(3)
s = Square(3)
r = Rectangle(3, 2)

@show combined_area(c, s)
@show combined_area(s, r);
```

Çıktı:

```
combined_area(c, s) = 16.068583470577035
combined_area(s, r) = 15.0
```

Soyut tiplerin polimorfik bir biçimde kullanılmasına rağmen, Julia gerekli olan çeşitli tipler için etkin kod derlemeyi sürdürme kapasitesine sahiptir.

7.0.2 Kompozisyon ve Metot Yönlendirme: Kalıtıma Bir Alternatif

Julia'da sınıfların eksikliği hakkında yaygın bir şikayet, yapıların (structs) üst sınıflardan veri alanlarını (data fields) miras alamamasıdır. Kalıtımın (özellikle veri düzeninin) aşırı kullanımının karmaşık kodlara yol açma eğiliminde olduğu görüşü yaygındır, ancak elbette faydalı olduğu durumlar da mevcuttur. Julia'da kalıtım kullanılmak istendiğinde, aynı alan adının tüm yapılarda tekrar tekrar yazılması gerekebilir. Bu durumun daha açık bir yaklaşım sunduğu düşünülmektedir, ancak bu kişisel tercihlere göre değişebilir.

Her durumda, kompozisyon genellikle kalıtımdan daha üstün bir desen olarak kabul edilir, çünkü en azından kısmen daha açık bir yapı sunar. Kompozisyonel kalıtım stratejisi, gerekli arayüze sahip tipi bir üst tip yerine bir alan olarak içeren yapılar oluşturmak ve ardından uygun metotları isteğe bağlı olarak bu alana yönlendirmektir. Julia, eval fonksiyonu ile bunu kolaylaştırır, zira metotları bir döngü ile yönlendirmek mümkündür. Julia'da bu tür önişlemler için eval kullanmak tipik bir yaklaşımdır.

```
struct HasInterestingField
    data::String
end

double(hif::HasInterestingField) = hif.data ^ 2
shout(hif::HasInterestingField) = uppercase(string(hif.data, "!"))

#kompozisyonel yol bu alanları başka bir yapıya eklemektir.
struct WantsInterestingField
    interesting::HasInterestingField
    WantsInterestingField(data) = new(HasInterestingField(data))
end

# forward methods
for method in (:double, :shout)
    @eval $method(wif::WantsInterestingField) = $method(wif.interesting)
end

```

```
#aynılardır:
#     double(wif::WantsInterestingField) = double(wif.interesting)
#     shout(wif::WantsInterestingField) = shout(wif.interesting)
wif = WantsInterestingField("foo")
@show shout(wif)
@show double(wif);
```

Çıktı:

```
shout(wif) = "FOO!"
double(wif) = "foofoo"
```

eval fonksiyonu metotları yönlendirir ve iç yapıcı (inner constructor), WantsInterestingField kullanıcılarının, üyelerinden birinin HasInterestingField olduğunu önemsememesini sağlar. Eğer mevcut bir HasInterestingField örneği ile WantsInterestingField örneği oluşturmanın faydalı olabileceği düşünülseydi, bu bir dış yapıcı (outer constructor) olabilir veya olmalıydı. eval kullanmanın yanı sıra, isteğe bağlı metotların yönlendirilmesi için bir makro da yazılabilir. Aşağıda, ilgili değeri ilk argüman olarak alan metotları yönlendiren bir makro örneği sunulmuştur:

```
#"Bir yapıya (struct) ait metotların, ilgili yapının bir veri alanına yönlendirilmesi işlemi.
#Bu durum, oluşturduğunuz kompozisyon için faydalıdır.
#Sözdizimsel olarak: @forward CompositeType.property Base.iterate Base.length :*
#Sembol değişmezlerinin Base.:symbol formuna otomatik dönüşümü, özellikle ekleme operasyonları için avantajlıdır.
#Dahili operatörlere metot eklemek için.@forward makrosunun imzası şu şekildedir: macro forward(property, functions...)
    structname = property.args[1]
    field = property.args[2].value
    block = quote end
    for f in functions
        #Operatörler için bu senaryo,
        if f isa QuoteNode
            f =:(Base. (f.value))def1=:(f(x::structname,y)=f(x.field,y))
               def2=:(f(x,y::structname)=f(x,y.$field))
```

```
            push!(block.args, def1, def2)

        #Kalan durumlar açısından
        else
            def = :(f(x::structname, args...; kwargs...) = f(x.field, args
                ...; kwargs...))
            push! (block.args, def)
        end
    end
    esc(block)
end
#Uygulamalı Gösterim:
struct Foo
    x::String
end
#Foo tipinin x alanına Base.string, tüm çarpma (:*), ve üs alma (:^) operat
   örleri yönlendiriliyor.
@forward Foo.x Base.string :* :^

foo = Foo("foo")

#string(foo, "!") çağrısı aslında string(foo.x, "!") olarak işlenir.
@show string(foo, "!")

#foo * "!" çağrısı aslında foo.x * "!" olarak işlenir.
@show foo * "!"

#foo ^ 4 çağrısı aslında foo.x ^ 4 olarak işlenir.
@show foo ^ 4;
```

Çıktı:

```
string(foo, "!") = "foo!"
foo * "!" = "foo!"
foo ^ 4 = "foofoofoofoo"
```

Bu kod, 2-clause BSD lisansı altında özel olarak kullanılan bir kütüphanenin parçasıdır ve herhangi bir garanti olmaksızın serbestçe kopyalanabilir. Ayrıca, popüler (veya "cool") Lazy.jl paketinde, birçok kişi tarafından kullanılan belgelenmemiş bir @forward makrosu da bulunmaktadır. Bu makronun yukarıda bahsedilenden daha sağlam olması muhtemeldir; zira yazarı Mike Innes, bu alanda bir dehadır.

Christopher Rackauckas'ın makalesi, Julia'da @def makrosu kullanılarak kompozisyona farklı bir yaklaşım sunmaktadır ve bu da incelenmeye değerdir.

7.0.3 Generikler: Statik Tip Güvenli Dinamik Tip Denetimi

Generikler, tam olarak dinamik tip denetiminin bir formu değildir. Ancak, OCaml ve Haskell gibi dillerde yaygın olarak kullanıldıklarında, statik tip denetimli dillerin sağladığı güvenlik garantileriyle dinamik olarak tip denetimliymiş gibi bir his yaratırlar. Julia'da generikler güvenlik için kullanılabilirken, aynı zamanda hem verimli hem de polimorfik yapılar oluşturmanın anahtarıdırlar. Point yapısına geri dönecek olursak, x ve y değerlerini temsil etmek için herhangi bir tipin kullanılması istenirse, ancak hala çıkarılabilir tiplerin verimliliği korunmak istenirse, bu noktada generikler devreye girer:

```
struct GenPoint{T}
    x::T
    y::T
end

GenPoint(1, 3)
```

Çıktı:

```
GenPoint{Int64}(1, 3)
```

Yukarıdaki kodda, küme parantezleri içinde bir tipi parametre olarak alan bir *tip yapıcı* oluşturulmuştur. Bu yapıcıdan daha sonra örnekler oluşturulabilir. Julia, yapıcı kullanılırken tip parametresinin atlanmasına izin verir, çünkü bu parametre argümanlardan çıkarılabilir. Bu özel durumda, tip yapıcısının yalnızca bir jenerik parametresi olduğu için x ve y aynı tipte olmalıdır.

```
GenPoint(1, 3.0)
```

Çıktı:

```
MethodError: no method matching GenPoint(::Int64, ::Float64)
Closest candidates are:
  GenPoint(::T, !Matched::T) where T at In[5]:2

Stacktrace:
 [1] top-level scope at In[6]:1
```

Görüldüğü gibi, her iki argüman da aynı tipte olmalıdır. Ancak, tip yapıcıları ek jenerik parametrelere sahip olabilir:

```
struct GenPoint2{X,Y}
    x::X
    y::Y
end

GenPoint2(1, 3.0)
```

Çıktı:

```
GenPoint2{Int64,Float64}(1, 3.0)
```

Bir noktanın x ve y koordinatlarının aynı tipte olması iyi bir fikir gibi görünse de, ilk genel nokta implementasyonunda hala bir sorun bulunmaktadır – bu implementasyon belki de *çok* geneldir.

```
GenPoint("foo", "bar")
```

Çıktı:

```
GenPoint{String}("foo", "bar")
```

Dizeler içeren bir noktanın anlamsız olduğu ve x ve y'ye uygulanabilecek işlemlerle ilgili her türlü varsayımı bozacağı açıktır. Bu durumda ne yapılabilir?

```
struct RealPoint
```

```
    x::Real
    y::Real
end

RealPoint(0x5, 0xaa)
```

Çıktı:

```
RealPoint(0x05, 0xaa)
```

Burada, alanların tipi olarak Real soyut tipi kullanılmıştır. Bu yaklaşım işe yarasa da, aslında bu şekilde yapılmaması önerilir. Real somut bir tip olmadığı için Julia derleyicisi RealPoint'in veri düzeni hakkında herhangi bir varsayımda bulunamaz ve metot seçimi derleme zamanı yerine çalışma zamanında gerçekleşmek zorunda kalır. Bunun doğru yolu aşağıdaki gibidir:

```
struct Point{T<:Real}
    x::T
    y::T
end

@show Point(1, 3)
@show Point(1.4, 2.5);
```

Çıktı:

```
Point(1, 3) = Point{Int64}(1, 3)
Point(1.4, 2.5) = Point{Float64}(1.4, 2.5)
```

Yukarıdaki örnekte, jenerik T'nin Real tipinin bir alt tipi olması gerektiği belirtilmiştir. Bu sayede aşağıdaki gibi anlamsız değerler engellenebilir:

```
Point("foo", "bar")
```

Çıktı:

```
MethodError: no method matching Point(::String, ::String)
```

```
Stacktrace:
 [1] top-level scope at In[11]:1
```

Not: Soyut tiplerin jenerik alt tiplerini kullanmak (yukarıdaki gibi), yapılar içinde alanların tiplerini belirtirken iyi performans için gereklidir, ancak metot imzalarında gerekli değildir. Yapıların doğru bir şekilde tanımlanması ve tip istikrarına dikkat edilmesi durumunda, derleyici her şeyin tipini bilir. Bir metodun tiplerine optimizasyon için asla güvenmez.

Generikler, özellikle konteynerler (containers) uygulamak için kullanışlıdır. Bir bağlı liste (linked list) örneği ile bu durum incelenecektir. İlk olarak kod, ardından açıklamalar sunulmuştur:

```
struct Nil end

struct List{T}
    head::T
    tail::Union{List{T}, Nil}
end
# Bir diziden liste oluşturma fonksiyonu
mklist(array::AbstractArray{T}) where T =
    foldr(List{T}, array, init=Nil())
# İterasyon protokolünün uygulanması
Base.iterate(l::List) = iterate(l, l)
Base.iterate(::List, l::List) = l.head, l.tail
Base.iterate(::List, ::Nil) = nothing
# Uygulamalı gösterim:
list = mklist(1:3)
@show list
for val in list
    println(val)
end
```

Çıktı:

```
list = List{Int64}(1, List{Int64}(2, List{Int64}(3, Nil())))
```

```
1
2
3
```

Yukarıdaki kodda, ilk olarak listenin sonunu işaret eden Nil tipi tanımlanmıştır.

Ardından, 3-6. satırlarda bir listedeki düğümün T tipinde bir başlık (head) ve aynı tipte başka bir düğüm veya nothing olabilecek bir kuyruk (tail) içermesi gerektiği belirtilmiştir.

ve 10. satırlar, bir diziden (array) bir liste oluşturan bir fonksiyonu tanımlar. Generikler açısından tek ilginç kısım, 9. satırdaki fonksiyon bildirimidir, burada jenerik T fonksiyon imzasında bildirilmiştir:

```
mklist(array::AbstractArray{T}) where T
```

Generikler, fonksiyon gövdelerinde bu şekilde kullanılabilir. Dizinin eleman tipini fonksiyon gövdesinde bir değere dönüştürür. Bu tip daha sonra kullanılır. Örneğin:

```
1 mklist([1, "foo"])
```

Çıktı:

```
List{Any}(1, List{Any}("foo", Nil()))
```

Aksi takdirde, başlangıç listesi değerinin tipi olan String'e varsayılan olarak ayarlanacağı ve üzerine bir Int64 düğümünün örneklendirilemeyeceği için bu durum çalışmazdı. Bu strateji typeof fonksiyonunu uygulamak için de kullanılabilir.

```
1 whichtype(::T) where T = T
2 whichtype("foo")
```

Çıktı:

```
String
```

Ancak, bu kadar kolay yapılabilmesi, typeof'u REPL dışında nadiren kullanma ihtiyacı olduğu anlamına gelir.

13-15. satırlar, tipimiz için Base.iterate'e metotlar ekleyerek iterasyon protokolünü uygular. Iterasyon protokolünü anlamak için bu bağlantı incelenmelidir. 14. satırda, Base.iterate(::List, l::List) ifadesinde ilk argümana bir isim verilmemiştir, çünkü değeri asla kullanılmayacaktır. Bu sadece doğru gönderimin seçilmesini sağlamak içindir. Bu durum daha sonra trait deseni ile tekrar gündeme gelecektir. 15. satır aynı deseni terminal durumu uygulamak için kullanır. Eğer iterate'in state argümanı nothing ise, tüm düğümlerin tükendiği anlamına gelir ve iterasyonun bittiğini bildirmek için biz de nothing döndürürüz (yinelenebilir-durum 2-tuple yerine). Esasen, generikler Julia'da bu şekilde çalışır.

7.0.4 Trait Deseni

Traitler, belirli bir tipin doğru arayüzleri uygulayacağına dair derleyiciye bir nevi sözleşme niteliğindedir ve bu nedenle o arayüzü gerektiren herhangi bir fonksiyon için bir argüman olarak kullanılabilir. Bu, klasik kalıtıma göre daha üstün bir alternatif olarak kabul edilmektedir. Normalde bir sınıfın yalnızca bir üst tipi varken, bir tipin birçok trait'i uygulayabilmesi mümkündür. Evet, çoklu kalıtım (multiple inheritance) mevcuttur, ancak bu genellikle karmaşık kodlara yol açarken, traitler bu karmaşıklığı önlemeye yardımcı olma eğilimindedir. Rust'taki traitler, Go'daki arayüzler ve Haskell'deki tip sınıfları, farklı dillerde davranışsal alt tiplemeye yönelik bu yaklaşımların örnekleridir.

Julia'da kalıtım olmaksızın davranışsal alt tipleme dil düzeyinde bir özellik değildir, ancak standart kütüphanede sıkça karşılaşılan, bazen deseni öneren Julia katkıcısı Tim Holy'nin adını taşıyan "Holy trait" olarak anılan popüler bir desen bulunmaktadır.

Bu desen hakkında resmi belgelerde buradan ve orijinal olarak ortaya çıktığı github tartışmasından daha fazla bilgi edinilebilir. Standart kütüphanedeki traitlerden birini kullanmak gerekebilecek bir alan, kendi yinelenebilir (iterable) tipini uygularken ortaya çıkar. Base.IteratorSize adı verilen bir trait sınıfı vardır ve bu, yinelenebilir nesnenin boyutu hakkında ipuçları sağlamak için kullanılabilir, böylece yayın (broadcasting) ve eşleme (mapping) gibi işlemler sırasında daha verimli alan tahsisi yapılabilir. Örneğin, yukarıdaki bağlı liste implementasyonunda listenin uzunluğunu belirlemek için bir metot bulunmadığından, Base.SizeUnknown trait'i eklenebilir:

```
Base.IteratorSize(::List) = Base.SizeUnkown()
```

Ancak, bir uzunluk fonksiyonu da uygulanabilir:

```
Base.length(l::List) = 1 + length(tail(l))
Base.length(::Nothing) = 0
#Muhtemeldir ki, Nothing tipi için Base.length fonksiyonunun geçersiz kılı
    nması uygun bir yaklaşım değildir.
#Bunun yerine, List yapısı için kendi Nil tipimizi oluşturmak daha uygun
    bir tercih olabilirdi.

Base.IteratorSize(::List) = Base.HasLength()
```

Bu, daha sonra standart kütüphanedeki çeşitli fonksiyonlarda verimli tahsis için kullanılacaktır. Elbette, yalnızca uzunluğunu almak için listeyi dolaşmanın ne kadar verimli olabileceği hala bir soru işaretidir, ancak bu, performansa duyarlı kodlarda bağlı listelerden kaçınılması gereken birkaç nedenden biridir. Bu sadece bir örnektir.

Standart kütüphane belgeleri, sağladığı traitlerin nasıl kullanılacağını etkili bir şekilde göstermekle birlikte, kendi traitlerini nasıl oluşturulacağını göstermekten garip bir şekilde geri kalmaktadır. En basit haliyle, bir trait, yalnızca boş bir yapı ve en genel durumda hata veren bir gönderimden ibarettir. Daha sonra, bir metot gönderiminde yapıcıyı kullanarak trait kontrol edilir.

```
struct Zlurmable end
Zlurmable(::T) where T = error("Type $T doesn't implement the Zlurmable
    trait")

zlurm(x) = zlurm(Zlurmable(x), x)
zlurm(::Zlurmable, x) = x + 1
```

```
zlurm (generic function with 2 methods)
```

Ancak, henüz Zlurmable trait'ine sahip bir tip bulunmadığından zlurm fonksiyonunu kullanmak mümkün değildir.

```
zlurm(1)
```

Çıktı:

```
Type Int64 doesn't implement the Zlurmable trait

Stacktrace:
 [1] Zlurmable(::Int64) at ./In[16]:2
 [2] zlurm(::Int64) at ./In[16]:4
 [3] top-level scope at In[17]:1
```

Bu durumda, bir tipe trait eklemek gerekmektedir:

```
Zlurmable(::Int64) = Zlurmable()
zlurm(3)
```

Çıktı:

```
4
```

Julia'daki traitler, esasen tipinizin doğru arayüzleri uygulayacağına dair (zorunlu olmayan) bir vaattir. Julia kodu varsayılan olarak dinamik olarak tip denetimli olduğu için, bu konuda umursamaz bir yaklaşım benimsenebilir ve fonksiyon doğrudan jenerik olarak yazılabilir, traitler gibi güvenlikler önemsenmeyebilir. Ancak, kalıtım karışıma eklendiğinde belirgin faydalar ortaya çıkar. Bu, trait arayüzünü uygulamanın farklı yollarına izin verir ve metotların farklı trait tiplerine göre optimize edilmesini sağlar. Kendi Base.IteratorSize versiyonu uygulanacak olsaydı, şuna benzerdi (aslında, tam olarak şuna benzer):

```
abstract type IteratorSize end
struct SizeUnknown <: IteratorSize end
struct HasLength <: IteratorSize end
struct HasShape{N} <: IteratorSize end
```

```
5 struct IsInfinite <: IteratorSize end
6
7 IteratorSize(::Any) = HasLength()
```

Çıktı:

```
IteratorSize
```

Burada, bir yineleyicinin boyutu hakkında düşünmek için çeşitli yollar bulunmaktadır ve farklı gönderimler bu bilgiden en iyi şekilde yararlanabilir. Her nedense, varsayılan IteratorSize HasLength olarak ayarlanmıştır. Neden böyle bir seçim yapıldığı belirsizdir, zira bu, yineleyicinin bir uzunluk fonksiyonu uyguladığını varsayar ve bulamadığında hata verir; oysa IteratorSize(::MyType)'ı dört seçenekten biriyle uygulamanız gerektiğini bildirmez. Boyutun bilinmediği birçok durum bulunmaktadır (örneğin, G/Ç ile uğraşırken).

8 MODÜLLER VE PAKET YÖNETİMİ

8.1 Paketler (Packages)

9 DOSYA İŞLEMLERİ VE UYGULAMALAR

9.1 Julia'da Metin Dosyalarının Okunması ve İşlenmesi (Reading and Manipulating Text Files)

9.2 Çizim (Plotting)

9.3 Temel Doğrusal Cebir (Basic Linear Algebra)

9.4 Faktörizasyonlar (Factorizations)

10 HATA AYIKLAMA VE PROGRAMLAMA SÜRECİ

10.1 İlk Julia Kodunun İncelenmesi

10.2 Hata Ayıklama ve Hata Mesajları

10.3 Programlama: Uygulanmış Biçimsel Mantık

Kaynaklar

[1] Julia Documentation. (n.d.). *Julia Programming Language*. Erişim adresi: https://docs.julialang.org/en/v1/

[2] GLCS. (n.d.). *Install Julia and VS Code*. Erişim adresi: https://blog.glcs.io/install-julia-and-vscode

[3] Arslan, İ. (n.d.). *Yeni bir programlama dili: Julia*. Medium. Erişim adresi: https://ilker-arslan.medium.com/yeni-bir-programlama-dili-julia-31f5e871339b

[4] Julia Data Science. (n.d.). *Syntax*. Erişim adresi: https://juliadatascience.io/syntax

[5] ZetCode. (n.d.). *Julia string tutorial*. Erişim adresi: https://zetcode.com/julia/string/

[6] MatecDev. (n.d.). *Julia Structs*. Erişim adresi: https://www.matecdev.com/posts/julia-structs.html

[7] Functional Noise. (2021). *Julia Encapsulation*. Erişim adresi: https://www.functionalnoise.com/pages/2021-07-02-julia-encapsulation/

[8] Aaron, N. (n.d.). *OO and polymorphism in Julia*. GitHub. Erişim adresi: https://github.com/ninjaaron/oo-and-polymorphism-in-julia