Modern Bash Scripting >_

Ender KUŞ

Modern Bash Scripting

Ender KUŞ

Version 1.0, 2025-12

İçindekiler

Ön Bölümler

Yazar Hakkında

Merhaba, ben Ender KUŞ.

Yazılım geliştirme serüvenime Ruby on Rails ve PHP geliştiricisi olarak başladım. Hem yurt içinde hem de yurt dışında çeşitli firmalarda, farklı ölçeklerdeki projelerde yer aldım. Bu süreçte sadece uygulama kodu yazmakla kalmadım, o uygulamaların üzerinde koştuğu işletim sistemi katmanında da bolca ter döktüm.

Daha önce Kodlab yayınlarından çıkan **Ruby Programlama** kitabını sizlerle buluşturmuştum. Yazma ve paylaşma tutkum, beni bu sefer de Linux ve Bash dünyasının derinliklerine götürdü.

Şu anda Türkiye'nin önde gelen özel bir şirketinde Linux, İzleme (Monitoring) ve Otomasyon Takım Lideri olarak görev yapıyorum. Ekibimle birlikte karmaşık sistemleri yönetiyor, süreçleri otomatikleştiriyor ve altyapıların sağlıklı çalışmasını sağlıyoruz.

Bu kitapta, yıllar içinde edindiğim tecrübeleri ve "keşke yolun başındayken bilseydim" dediğim incelikleri samimi bir dille size aktarmaya çalıştım. Umarım bu satırlar, sizin terminal yolculuğunuzda da iyi bir rehber olur.

Önsöz: Neden Bash Script?

Linux'ta aynı sıkıcı işleri tekrar tekrar yapmaktan, saatlerinizi heba etmekten yorulmadınız mı? O zaman bu kitap tam size göre!

Modern Bash Scripting, o sıkıcı görevleri otomatize etmeniz için ihtiyacınız olan tüm yetenekleri size kazandıracak. Bu kitabı okuduktan sonra Linux üzerinde çalışırken çok daha verimli olacaksınız. Ama daha da önemlisi; size söz veriyorum, geceleri artık daha fazla uyuyabileceksiniz! (Çünkü işleri script'ler yapacak.)

Kitabımız, temel düzeyde Linux bilgisine sahip olduğunuzu ve komut satırına aşina olduğunuzu varsayar. Ancak endişelenmeyin, giriş bölümlerinde hafızanızı tazeleyeceğiz.

Bu Kitap Nasıl Kullanılmalı?

Bu kitaptan maksimum faydayı sağlamak istiyorsanız, kronolojik bir sıra izlemelisiniz. Çünkü her bölüm, sizi bir sonraki bölüme hazırlar. Bu yüzden maceraya birinci bölümden başlamanızı ve finiş çizgisine ulaşana kadar adım adım, bölüm bölüm ilerlemenizi tavsiye ederim.

Ayrıca, sadece okuyucu olmayın, uygulayıcı olun! Kitapta gördüğünüz her bir bash script'ini kendi ellerinizle yazmalı ve çalıştırmalısınız. Bir bash script'ini sadece okuyup geçmeyin; klavyenizin başına geçin ve o kodu çalıştırın!

Yasal Uyarı ve Sorumluluk Reddi

Bu kitapta yer alan kodlar ve teknik bilgiler eğitim amaçlıdır. Komutları denerken

oluşabilecek veri kayıplarından veya sistem hatalarından yazar sorumlu tutulamaz.

Scriptleri denerken lütfen test ortamları (Sanal Makine, Docker vb.) kullanın ve kritik sistemlerde çalıştırmadan önce mutlaka **yedek alın**. Tüm sorumluluk size aittir.

Chapter 1. Temeller ve Giriş

1.1. Bash Nedir ve Neden Kullanılır?

Bash (Bourne Again SHell), Linux ve Unix dünyasının en popüler komut satırı kabuğudur (shell). İsmi, Unix'in ilk kabuklarından biri olan "Bourne Shell"in (sh) geliştiricisi Stephen Bourne'a bir saygı duruşu niteliğindedir ve kelime oyunu içerir ("Bourne Again" - "Born Again"). 1989 yılında Brian Fox tarafından GNU Projesi için yazılmıştır.

Peki, "kabuk" (shell) derken neyi kastediyoruz? Kabuk, sizinle (kullanıcı) işletim sisteminin çekirdeği (kernel) arasında bir köprü görevi gören, metin tabanlı bir arayüzdür. Siz klavyeden komutlar girersiniz, kabuk bu komutları yorumlar ve çekirdeğe iletir. Çekirdek işlemi yapar, sonucu kabuğa gönderir, kabuk da size gösterir.

Neden Bash Kullanmalıyız?

1. **Her Yerdedir:** Hemen hemen her Linux dağıtımında, macOS'ta ve artık Windows (WSL) üzerinde varsayılan veya kolayca erişilebilir durumdadır. Bir sunucuya bağlandığınızda sizi karşılayacak olan şey muhtemelen Bash'tir.
2. **Otomasyonun Kalbidir:** Sistem yöneticileri ve geliştiriciler için hayat kurtarıcıdır. Her gün elle yaptığınız o sıkıcı dosya yedekleme işlemini, log analizini veya sunucu güncellemesini tek bir Bash script'i ile otomatik hale getirebilirsiniz.
3. **Yapıştırıcıdır:** Bash, farklı programları birbirine bağlama konusunda ustadır. `grep`, `awk`, `sed` gibi güçlü araçları Bash ile birleştirerek harikalar yaratabilirsiniz.
4. **Hız ve Verimlilik:** Grafik arayüzler (GUI) bazen yavaştır veya her işi yapamaz. Komut satırı ise ışık hızındadır ve size sistem üzerinde tam kontrol sağlar.

Kısacası Bash öğrenmek, bilgisayarınızla konuşmanın en doğrudan ve en güçlü yolunu öğrenmektir.

1.2. Terminal Temelleri

Bash script yazmaya başlamadan önce, komut satırında kendimizi evimizde gibi hissetmeliyiz. "Ben zaten bunları biliyorum" diyorsanız harika, hızlıca bir hafıza tazelemesi yapıp geçebiliriz.

Terminali açtığınızda sizi genellikle şöyle bir satır karşılar:

```
kullanici@bilgisayar:~$
```

Buradaki `$` işareti (bazen `#` olabilir, eğer root iseniz), Bash'in sizden komut beklediğini gösterir. `~` (tilde) işareti ise o an ev dizininizde (home directory) olduğunuzu belirtir.

İşte script yazarken sıkça kullanacağımız, adınız gibi bilmeniz gereken o temel komutlar:

- `pwd` **(Print Working Directory):** "Neredeyim?" sorusunun cevabıdır. O an dosya sisteminin hangi klasöründe olduğunuzu tam yol (path) olarak gösterir.

- `ls` **(List):** Bulunduğunuz klasördeki dosya ve klasörleri listeler.
- `ls -l`: Detaylı listeleme (izinler, boyut, tarih vb.).
- `ls -a`: Gizli dosyaları (baştan nokta ile başlayanlar) da gösterir.
- `cd` **(Change Directory):** Klasörler arası ışınlanma aracınızdır.
- `cd Documents`: Belgeler klasörüne girer.
- `cd ..`: Bir üst klasöre çıkar.
- `cd ~` (veya sadece `cd`): Doğrudan ev dizininize döner.
- `cd -`: Bir önceki bulunduğunuz klasöre geri döner (TV kumandasındaki "Last Channel" tuşu gibi).
- `mkdir` **(Make Directory):** Yeni bir klasör oluşturur.
- `mkdir projelerim`
- `touch`**:** Boş bir dosya oluşturur veya var olan dosyanın zaman damgasını günceller.
- `touch notlar.txt`
- `rm` **(Remove): Dikkat!** Dosya veya klasör siler. Geri dönüşüm kutusu yoktur, giden gider.
- `rm dosya.txt`: Dosyayı siler.
- `rm -rf klasör`: Klasörü ve içindeki her şeyi zorla siler (Çok dikkatli olun!).
- `man` **(Manual):** Her şeyin kullanım kılavuzu. `man ls` yazarak `ls` komutunun tüm detaylarını öğrenebilirsiniz.

Bu komutlar, Bash ile dans ederken atacağınız en temel adımlardır.

1.3. İlk Script: "Merhaba Dünya"

Yazılım dünyasının değişmez geleneğiyle başlayalım: Ekrana "Merhaba Dünya" yazdırmak. Ancak bunu yaparken, Linux dünyasının efsanevi (ve bazen korkutucu bulunan) editörü **vi**'yi kullanacağız. Korkmayın, sadece hayatta kalmanıza yetecek kadarını anlatacağım.

Bir script dosyası oluşturmak için `vi` editörünü şöyle çağırıyoruz:

```
vi merhaba.sh
```

Bu komutu yazdığınızda karşınıza boş bir ekran gelecek. Şu an **Komut Modu**'ndasınız (Command Mode). Yani klavyeye bastığınızda harf yazmaz, komut verirsiniz. Yazı yazmak için **Ekleme Modu**'na (Insert Mode) geçmeliyiz.

1. Klavyeden `i` tuşuna basın. (Ekranın sol alt köşesinde `-- INSERT --` yazısını görebilirsiniz).
2. Artık yazabiliriz. Şu satırı yazın:

```
echo "Merhaba Dünya"
```

Bu işimizi görecektir. Şimdi dosyayı kaydedip çıkmamız lazım.

1. Önce `ESC` tuşuna basarak Ekleme Modu'ndan çıkın (`-- INSERT --` yazısı kaybolmalı).
2. Ardından sırasıyla `:` (iki nokta üst üste), `w` (write - kaydet) ve `q` (quit - çık) tuşlarına basın. Ekranın sol alt köşesinde `:wq` göreceksiniz.
3. `Enter` tuşuna basın.

Tebrikler! İlk dosyanızı `vi` ile oluşturdunuz. Şimdi bu şaheseri çalıştıralım:

```
bash merhaba.sh
```

Eğer her şey yolunda gittiyse, terminalde şunu görmelisiniz:

```
Merhaba Dünya
```

Hoş geldiniz! Artık resmen Bash script dünyasına adım attınız.

1.4. Shebang (#!) Nedir?

Bir önceki örnekte dosyanın içine sadece `echo "Merhaba Dünya"` yazdık ve onu `bash merhaba.sh` diyerek çalıştırdık. Yani Linux'a "Bu dosyayı al ve Bash yorumlayıcısı ile çalıştır" emrini verdik.

Ancak profesyonel dünyada script'ler genellikle doğrudan isimleriyle (örneğin `./merhaba.sh`) çalıştırılır. Linux'un bu dosyanın bir Bash script'i olduğunu anlayabilmesi için dosyanın **en başına** özel bir satır eklemeliyiz. İşte buna **Shebang** diyoruz.

Shebang, `#` (sharp) ve `!` (bang) karakterlerinin birleşiminden oluşur ve ardından yorumlayıcının yolu gelir:

```
#!/bin/bash
```

Bu satır, işletim sistemine şunu söyler: "Hey, bu dosyanın geri kalanını `/bin/bash` programını kullanarak çalıştır!"

Eğer Python script'i yazsaydık `!/usr/bin/python3`, `Perl yazsaydık!/usr/bin/perl` kullanacaktık.

Hadi script'imizi güncelleyelim. `vi merhaba.sh` ile dosyayı açın ve en başa şu satırı ekleyin:

```
#!/bin/bash
echo "Merhaba Dünya"
```

Artık script'imiz kimliğini kazandı. Ancak henüz "kendi başına" çalışmaya hazır değil. Neden mi? Cevabı bir sonraki başlıkta.

1.5. Çalıştırma İzinleri ve PATH Kavramı

Shebang satırını ekledikten sonra script'i şu şekilde çalıştırmayı deneyin:

```
./merhaba.sh
```

Muhtemelen şu hatayı alacaksınız: `Permission denied` (Erişim engellendi). Linux'ta güvenlik gereği, oluşturduğunuz dosyalar varsayılan olarak "çalıştırılabilir" (executable) değildir. Bir dosyanın program gibi çalışabilmesi için ona **çalıştırma izni (+x)** vermelisiniz.

Bunu `chmod` (change mode) komutuyla yapıyoruz:

```
chmod +x merhaba.sh
```

Şimdi tekrar deneyin:

```
./merhaba.sh
```

Ve bingo! "Merhaba Dünya" yazısını gördünüz.

Peki neden `./` koyuyoruz? Normalde `ls`, `git`, `vi` gibi komutları yazarken başlarına `./` koymayız. Çünkü bu komutların bulunduğu klasörler (genellikle `/bin`, `/usr/bin` vb.), sistemin **PATH** adı verilen özel bir değişkeninde kayıtlıdır. Linux, bir komut yazdığınızda onu PATH içindeki klasörlerde arar.

Sizin oluşturduğunuz script şu anki klasörde duruyor ve Linux güvenlik sebebiyle (yanlışlıkla aynı isimli bir dosyayı çalıştırmamanız için) **mevcut klasörü (current directory) PATH içinde saymaz.**

Bu yüzden Linux'a "Gitme, uzağa bakma; script tam olarak burada, bu klasörün (`.`) içinde!" demek için dosya yolunu açıkça belirtiriz: `./merhaba.sh`.

1.6. Yorum Satırları ve Kod Düzeni

Kod yazmak sadece bilgisayarlar için değil, insanlar (ve gelecekteki siz) içindir. 6 ay sonra kendi script'inizi açtığınızda "Ben burada ne yapmaya çalışmışım?" diye boş gözlerle bakmak istemiyorsanız, **yorum satırlarını** kullanın.

Bash'te `#` karakterinden sonra gelen her şey yorum olarak kabul edilir ve çalıştırılırken görmezden gelinir (Shebang hariç, o özeldir çünkü ilk satırdadır).

```
#!/bin/bash

# Bu script ekrana selamlama mesajı yazar.
# Yazan: Ender Kuş
# Tarih: 2025-12
```

```
echo "Merhaba Dünya" # Burası ekrana çıktı verir
```

Kod Düzeni İpuçları: 1. **Başlık Ekleyin:** Script'in ne işe yaradığını, kimin yazdığını en başa not düşün. 2. **Girinti (Indentation) Kullanın:** İleride `if` veya `while` blokları yazarken, kodun okunabilir olması için içerideki satırları boşluk veya tab ile içeriden başlatın. Okunabilirlik, hatasız kodun anahtarıdır.

Chapter 2. Değişkenler ve Veri Tipleri

2.1. Değişken Tanımlama ve Kullanma

Programlamanın ve script yazmanın en temel yapı taşı değişkenlerdir. Değişkenleri, verileri (bir isim, bir dosya yolu, bir sayı) geçici olarak hafızada tuttuğumuz kutucuklar gibi düşünebilirsiniz.

Bash'te değişken tanımlamak oldukça basittir, ancak çok katı bir kural vardır: **Eşittir işaretinin sağında ve solunda boşluk olmamalıdır!**

Yanlış:

```
isim = "Ender"   # HATA! Boşluk olmayacak.
yas = 30         # HATA!
```

Doğru:

```
isim="Ender"
yas=30
```

Değişken İsimlendirme Kuralları Bash, değişken isimleri konusunda bazı kurallara sahiptir. Hata almamak için bunlara uymalısınız:

1. **Harf, Sayı ve Alt Çizgi:** Değişken isimleri sadece harf (`a-z`, `A-Z`), sayı (`0-9`) ve alt çizgi (`_`) içerebilir.
2. **Sayı ile Başlayamaz:** bir değişken ismi sayı ile başlayamaz. `1inci_ders` hatalıdır, `ders_1` geçerlidir.
3. **Büyük/Küçük Harf Duyarlıdır:** `ISIM`, `isim` ve `Isim` üç farklı değişkendir.
4. **Özel Karakter Yok:** `!`, `*`, `-`, `?` gibi karakterler isimlendirmede kullanılamaz. (Örn: `benim-adım` yanlıştır, `benim_adim` doğrudur).
5. **Gelenek:** Genellikle sistem değişkenleri (PATH, HOME) BÜYÜK HARFLE, kendi tanımladığımız değişkenler küçük_harf veya kucukHarf (camelCase) ile yazılır.

Değişkeni Kullanmak Tanımladığımız değişkenin içindeki değeri okumak için başına `$` (dolar) işareti koyarız.

```
echo $isim
echo $yas
```

Daha güvenli ve temiz bir kullanım için değişken ismini süslü parantez içine almak (`${isim}`) iyi bir alışkanlıktır. Bu, değişkenin nerede bittiğini Bash'e açıkça gösterir.

```
echo "Benim adım ${isim} ve yaşım ${yas}."
```

Bu yaptığımız işleme programlama literatüründe **String Interpolation** (Metin İçine Değişken Gömme) denir. Çift tırnak kullandığımız sürece, Bash içerideki değişkenleri (${...}) algılar ve yerlerine gerçek değerlerini yazar.

Eğer süslü parantez kullanmazsak ne olur?

```
kelime="Bash"
echo "$kelimeing"
# Bash burada 'kelimeing' adında bir değişken arar, bulamaz ve boş basar.
# Doğrusu: echo "${kelime}ing" -> Bashing
```

Değişkeni Silmek (Unset) Tanımladığınız bir değişkenle işiniz bittiğinde veya onu hafızadan tamamen silmek istediğinizde `unset` komutunu kullanabilirsiniz.

```
isim="Mehmet"
echo $isim  # Çıktı: Mehmet

unset isim
echo $isim  # Çıktı: (Boş bir satır, çünkü değişken artık yok)
```

2.2. Veri Tipleri (String, Integer vb.)

Bash, C veya Java gibi "strongly typed" (sıkı tipli) bir dil değildir. Bash'te değişkenlerin "tipi" (türü) o kadar belirgin değildir; aslında Bash için hemen hemen her şey bir **String**'dir (metin).

String (Metin) En sık kullanacağınız veri tipidir.

```
mesaj="Merhaba Dünya"
dosya_yolu="/home/ender/belgeler"
```

Integer (Tamsayı) Bash, değişkenlere sayı atadığınızda onları saklar, ancak matematiksel işlem yapmadığınız sürece onları da metin gibi görür.

```
sayi=100
echo $sayi
```

Tırnak İşaretlerinin Sırrı (Quotes) Bash'te "Çift Tırnak" (") ve 'Tek Tırnak' (') arasında hayati bir fark vardır:

1. **Çift Tırnak ("):** İçindeki değişkenleri yorumlar.
2. **Tek Tırnak ('):** İçindekine dokunmaz, ne görüyorsa onu basar (Literal string).

Örnekle görelim:

```
isim="Ali"

echo "Merhaba $isim"  # Çıktı: Merhaba Ali
echo 'Merhaba $isim'  # Çıktı: Merhaba $isim
```

Eğer değişkenlerin değerini görmek istiyorsanız çift tırnak, içinde `$`, `!` gibi özel karakterler geçen bir metni (örneğin bir şifre) olduğu gibi korumak istiyorsanız tek tırnak kullanın.

2.3. Sabit (Read-only) Değişkenler

Bazen bir script içinde tanımladığınız bir değerin, programın ilerleyen kısımlarında kazara değiştirilmesini istemezsiniz. Örneğin, script'in çalıştığı ana dizin veya bir API anahtarı gibi.

Bash'te bir değişkeni "sabit" (değiştirilemez) yapmak için `readonly` komutu kullanılır.

```
#!/bin/bash

readonly SIRKET_ADI="TechCorp"
TR="Türkiye"
readonly TR

echo "Şirket: $SIRKET_ADI"
echo "Ülke: $TR"

# Değiştirmeye çalışalım
SIRKET_ADI="NewCorp"
```

Eğer bu script'i çalıştırırsanız, son satırda şöyle bir hata alırsınız: `bash: SIRKET_ADI: readonly variable`

Bu, scriptinizin güvenliği ve tutarlılığı için harika bir önlemdir.

2.4. Komut Çıktısını Değişkene Atama (Command Substitution)

Bash'in en güçlü özelliklerinden biri budur! Bir Linux komutunun çıktısını (output) alıp bir değişkene atayabilir ve bunu script'inizde kullanabilirsiniz.

Bunun için iki yöntem vardır, ancak modern dünyada **birincisini** kullanmanızı şiddetle tavsiye ederim.

Yöntem 1: `$()` Kullanımı (Önerilen)

```
tarih=$(date +%F)
dosya_sayisi=$(ls | wc -l)
```

```
neredeyim=$(pwd)

echo "Bugün tarih: $tarih"
echo "Burada $dosya_sayisi adet dosya var."
echo "Şu an $neredeyim klasöründeyiz."
```

Bash, önce `$()` parantezinin içindeki komutu çalıştırır, çıkan sonucu alır ve değişkene yapıştırır.

Yöntem 2: Backticks ` ` Kullanımı (Eski) Klavyede genellikle noktalı virgülün orada veya `AltGr` ile yapılan, tırnak işaretine benzeyen ama yatık olan işarettir.

```
tarih=`date`
```

Bu yöntem hala çalışır ancak okunması zordur ve iç içe komut kullanırken (nested commands) başınızı ağrıtır. O yüzden `$()` ile yola devam edin.

2.5. Özel Bash Değişkenleri

Bash, siz daha script'inizi yazmaya başlamadan önce sizin için arka planda birçok değişkeni hazır eder. Bunlara **Ortam Değişkenleri (Environment Variables)** denir.

İşte en sık karşılaşacaklarınız:

- `$HOME`: O anki kullanıcının ev dizini (Örn: `/home/ender`).
- `$USER`: O anki kullanıcının adı.
- `$PWD`: Bulunulan dizin (pwd komutunun değişken hali).
- `$SHELL`: Hangi kabuğu kullandığınız.
- `$HOSTNAME`: Bilgisayarın ağ üzerindeki adı.
- `$UID`: Kullanıcı ID numarası (User ID). Özellikle script'in root (UID 0) olarak çalışıp çalışmadığını kontrol etmek için kullanılır.
- `$RANDOM`: 0 ile 32767 arasında rastgele bir sayı üretir.

Örnek:

```
#!/bin/bash

echo "Merhaba $USER, hoş geldin!"
echo "Ev dizinin: $HOME"
echo "Senin kullanici numaran: $UID"
echo "Rastgele bir sayı: $RANDOM"
```

Bu değişkenler, script'inizi kimin, nerede çalıştırdığına göre dinamik davranmasını sağlar. İlerleyen bölümlerde (özellikle karar yapıları ve döngülerde) bu değişkenleri kullanarak çok daha yetenekli script'ler yazacağız.

Chapter 3. Kullanıcı Etkileşimi ve Argümanlar

3.1. Script'e Argüman Gönderme

Şu ana kadar yazdığımız script'ler kendi içine kapanıktı; ne tanımladıysak onu yaptılar. Ancak gerçek hayatta script'lerimizin dış dünyadan bilgi almasını isteriz. Örneğin, "bu dosyayı yedekle" derken **hangi** dosyayı yedekleyeceğini script'e söylememiz gerekir.

Bunu yapmak için script'i çalıştırırken yanına değerler (argümanlar) yazarız: `./yedekle.sh rapor.txt`

Bash, bu gönderilen değerleri `$1`, `$2`, `$3` gibi özel değişkenlerde saklar.

- `$0`: Script'in kendi adı (Örn: `./yedekle.sh`)
- `$1`: İlk argüman
- `$2`: İkinci argüman
- ... ve böyle dokuza kadar gider.
- `$#`: Toplam kaç argüman gönderildiğinin sayısı.
- `$@`: Gönderilen tüm argümanların listesi.

Örnek Uygulama: Selamlama Scripti Hemen `selamla.sh` adında bir dosya oluşturalım:

```
#!/bin/bash

isim=$1
soyisim=$2

echo "Merhaba $isim $soyisim!"
echo "Dosya adı: $0"
echo "Toplam $# argüman aldım."
echo "Tüm argümanlar: $@"
```

Çalıştıralım:

```
./selamla.sh Ender Kuş 30 Yazılımcı
```

Çıktı:

```
Merhaba Ender Kuş!
Dosya adı: ./selamla.sh
Toplam 4 argüman aldım.
Tüm argümanlar: Ender Kuş 30 Yazılımcı
```

Eğer hiç argüman vermezseniz (`./selamla.sh`), `$isim` ve `$soyisim` boş olacağı için sadece "Merhaba !" yazar.

3.2. Kullanıcıdan Girdi Alma (`read` komutu)

Bazen veriyi script çalışırken kullanıcıya sormak istersiniz. "Adınız nedir?", "Devam etmek istiyor musunuz?" gibi interaktif durumlar için `read` komutunu kullanırız.

Temel Kullanım:

```
echo "Adınızı yazın:"
read isim
echo "Memnun oldum, $isim."
```

Burada script "Adınızı yazın:" dedikten sonra durur ve kullanıcı bir şey yazıp Enter'a basana kadar bekler. Yazılan değer `isim` değişkenine atanır.

Daha Pratik Kullanım (`-p` parametresi): `echo` ile soru sormak yerine `read -p` (prompt) ile soruyu ve beklemeyi tek satırda yapabiliriz.

```
read -p "Adınız nedir? " isim
echo "Merhaba $isim."
```

Gizli Girdi Alma (`-s` parametresi): Şifre gibi ekranda görünmesini istemediğiniz bilgiler için `-s` (silent) parametresi kullanılır.

```
read -p "Kullanıcı adı: " k_adi
read -s -p "Şifre: " sifre
echo "" # Alt satıra geçmek için
echo "Giriş yapılıyor..."
```

Bu yöntemle script'lerinizi interaktif ve kullanıcı dostu hale getirebilirsiniz.

3.3. Argüman Kontrolü ve Varsayılan Değerler

Script'inize beklediğiniz argümanların gönderilip gönderilmediğini kontrol etmek önemlidir. Eğer kullanıcı argüman göndermeyi unutursa script'iniz hata verebilir veya -daha kötüsü- yanlış çalışabilir.

Bunun için en pratik yöntem, **Varsayılan Değer (Default Value)** atamaktır. Bash'te bunu yapmak için çok şık bir sözdizimi vardır: `${DEGISKEN:-VARSAYILAN}`.

Kullanımı: `isim=${1:-Misafir}`

Bu satır şunu demek ister: "Eğer 1. argüman (`$1`) varsa, `isim` değişkenine onu ata. EĞER YOKSA (veya boşsa), `isim` değişkenine 'Misafir' değerini ata."

Örnek: Yedekleme Scripti (Gelişmiş) Diyelim ki bir dosyayı yedekleyen script yazıyoruz. Eğer kullanıcı dosya adı vermezse, varsayılan olarak "notlar.txt" dosyasını yedeklesin istiyoruz.

```
#!/bin/bash

dosya=${1:-notlar.txt}

echo "$dosya dosyası yedekleniyor..."
# (Burada yedekleme komutları olurdu, örneğin cp $dosya $dosya.bak)
```

Deneyelim:

1. Argüman vererek: `./yedekle.sh rapor.pdf`
    - Çıktı: `rapor.pdf dosyası yedekleniyor…`
2. Argüman vermeden: `./yedekle.sh`
    - Çıktı: `notlar.txt dosyası yedekleniyor…`

Bu yöntem, karmaşık `if` yapılarına girmeden (henüz öğrenmedik!) script'lerinizi daha güvenli ve hataya dayanıklı hale getirmenin en kısa yoludur.

Chapter 4. Diziler (Arrays)

4.1. Dizi Tanımlama

Programlamada bazen tek bir değişkende birden fazla veri tutmak isteriz. Örneğin, bir sınıftaki öğrencilerin listesi veya sunucu isimleri gibi. İşte bunun için **Diziler (Arrays)** kullanılır.

Bash'te dizi tanımlamak oldukça basittir. Değerleri parantez `()` içine alır ve aralarına **boşluk** koyarız.

Temel Sözdizimi:

```
iller=("İstanbul" "Ankara" "İzmir" "Antalya")
sayilar=(10 20 30 40 50)
```

Dikkat Edilmesi Gerekenler: 1. **Ayırıcı Boşluktur:** Elemanları ayırmak için virgül (`,`) değil, boşluk kullanılır. 2. **Boşluklu İsimler:** Eğer eleman kendi içinde boşluk içeriyorsa mutlaka tırnak içine alınmalıdır. * Yanlış: `sehirler=(New York San Francisco)` → 4 eleman olur. * Doğru: `sehirler=("New York" "San Francisco")` → 2 eleman olur.

İndeks Numarası ile Atama: Dilerseniz doğrudan belirli bir sıraya da eleman yerleştirebilirsiniz:

```
meyveler[0]="Elma"
meyveler[1]="Armut"
meyveler[5]="Muz" # Aradaki 2,3,4 boş kalır
```

4.2. Dizi Elemanlarına Erişim

Bir diziyi tanımladık, peki içindeki verilere nasıl ulaşacağız? Bash'te diziler "sıfır tabanlıdır" (zero-indexed), yani saymaya 0'dan başlarız.

Dizi elemanlarına erişirken süslü parantez `{}` kullanımı **zorunludur**.

Tek Bir Elemana Erişim:

```
iller=("İstanbul" "Ankara" "İzmir" "Antalya" "Bursa")

echo ${iller[0]}  # Çıktı: İstanbul
echo ${iller[1]}  # Çıktı: Ankara
```

Tüm Elemanları Yazdırmak: Dizinin tamamını görmek için indeks yerine `@` veya `*` karakterini kullanırız. Genellikle `@` tercih edilir çünkü elemanları ayrı ayrı stringler olarak korur.

```
echo ${iller[@]}
```

```
# Çıktı: İstanbul Ankara İzmir Antalya Bursa
```

Dizi Uzunluğunu (Kaç Eleman Var?) Öğrenmek: Dizinin başına `#` koyarak eleman sayısını alabiliriz.

```
echo "Toplam il sayısı: ${#iller[@]}"
# Çıktı: Toplam il sayısı: 5
```

Belirli Bir Aralığı Almak (Slicing - Dilimleme): Dizinin sadece bir kısmını almak istediğinizde `${dizi[@]:BAŞLANGIÇ:ADET}` formülünü kullanırız. Buradaki mantığı iyi oturtmak önemlidir:

- **BAŞLANGIÇ (Offset):** Hangi indisten başlanacak? (0, 1, 2...)
- **ADET (Length):** O noktadan itibaren sağa doğru kaç eleman alınacak?

Örnek üzerinde görelim:

```
# Dizimiz: ("İstanbul" "Ankara" "İzmir" "Antalya" "Bursa")
# İndeksler:   0          1        2         3        4

# 1. indisten başla (Ankara) ve 2 tane al (Ankara, İzmir)
echo ${iller[@]:1:2}
# Çıktı: Ankara İzmir

# 3. indisten başla (Antalya) ve sonuna kadar git (Adet belirtmezsen sonuna kadar
alır)
echo ${iller[@]:3}
# Çıktı: Antalya Bursa
```

Diziyi Ters Çevirmek (Reverse): Bash'te diziyi ters çeviren doğrudan bir komut (Python'daki `reverse()` gibi) **yoktur**. Ancak Linux'un gücünü kullanarak bunu yapabiliriz. En pratik yöntemlerden biri, `tac` (cat'in tersi) komutunu kullanmaktır.

```
# Diziyi satır satır yazdırıp, tac ile ters çeviriyoruz
printf '%s\n' "${iller[@]}" | tac
```

Not: Bu işlem dizinin orijinalini değiştirmez, sadece ekrana ters basar. İleride döngüler konusunu işlediğimizde bunu daha kalıcı algoritmalarla nasıl yapacağımızı göreceğiz.

4.3. Diziye Eleman Ekleme ve Silme

Diziler statik değildir; sonradan yeni şehirler ekleyebilir veya çıkarabiliriz.

Diziye Eleman Ekleme (`+=` Operatörü): Mevcut bir dizinin sonuna yeni elemanlar eklemek için `+=` operatörü kullanılır.

```
iller=("İstanbul" "Ankara")
iller+=("İzmir")          # Tek eleman ekleme
iller+=("Antalya" "Bursa") # Çoklu ekleme

echo ${iller[@]}
# Çıktı: İstanbul Ankara İzmir Antalya Bursa
```

Diziden Eleman Silme (`unset`): Tıpkı değişkenlerde olduğu gibi `unset` komutu kullanılır. Ancak burada bir incelik vardır.

```
# 1. indisteki elemanı (Ankara) silelim
unset iller[1]

echo ${iller[@]}
# Çıktı: İstanbul İzmir Antalya Bursa
```

Dikkat: `unset` ile bir elemanı sildiğinizde, o indeks **boşalır ama kaymaz**. Yani `iller[0]` İstanbul'dur, `iller[2]` İzmir'dir. `iller[1]` artık boştur (null). Dizinin indeksi yeniden sıralanmaz.

Eğer tamamen temiz, sıralı (1., 2. diye giden) bir dizi istiyorsanız, diziyi yeniden oluşturmalısınız: `iller=( "${iller[@]}" )`

4.4. Dizi Üzerinde İşlemler

Dizilerle sadece ekleme çıkarma değil, elemanların içeriğini değiştirme işlemleri de yapabiliriz.

Bul ve Değiştir (Search and Replace): Dizi içindeki belirli bir kelimeyi veya harfi değiştirmek için `${dizi[@]/aranan/yeni}` yapısını kullanırız.

```
meyveler=("Elma" "Armut" "Muz" "Kara Elma")

# İçinde "Elma" geçen her şeyi "Apple" yapalım
yeni_meyveler=( "${meyveler[@]/Elma/Apple}" )

echo ${yeni_meyveler[@]}
# Çıktı: Apple Armut Muz Kara Apple
```

- Tek eğik çizgi `/` kullanırsanız sadece ilk bulduğunu değiştirir.
- Çift eğik çizgi `//` kullanırsanız (örneğin `${dizi[@]//aranan/yeni}`) bulduğu **her şeyi** değiştirir.

Diziyi Kopyalamak: Bir diziyi başka bir diziye kopyalamak için:

```
kopya_dizi=( "${orijinal_dizi[@]}" )
```

Bu işlem, orijinal dizideki tüm elemanları alır ve yeni bir dizi oluşturur. İndeks boşluklarını (unset

kaynaklı) temizlemenin (re-indexing) en kolay yolu da budur.

Chapter 5. Aritmetik İşlemler

5.1. `let`, `expr` ve `(( ))` Kullanımı

Bash'te matematiksel işlem yapmanın birden fazla yolu vardır. Tarihsel gelişim içinde farklı yöntemler ortaya çıkmıştır, ancak günümüzde hangisini kullanmanız gerektiğini bilmek önemlidir.

1. Çift Parantez (Önerilen Yöntem) Modern Bash yazımında matematiksel işlemler için altın standart budur. Okunması kolaydır, değişkene `$` koymadan erişebilirsiniz ve C-tarzı işlemleri destekler.

```
sayi1=10
sayi2=5

((sonuc = sayi1 + sayi2))
echo "Toplama Sonucu: $sonuc"
```

Eğer sonucu doğrudan yazdırmak isterseniz `$(( ))` yapısını kullanabilirsiniz:

```
echo "Çarpma Sonucu: $(( sayi1 * sayi2 ))"
```

2. `let` Komutu Eski alışkanlıklardan biridir. Hala çalışır ancak `(( ))` kadar pratik değildir.

```
let sonuc=sayi1+sayi2
```

3. `expr` Komutu (Eski ve Yavaş) Bash'in ilk zamanlarından kalma, harici bir programdır. Operatörlerle değişkenler arasında boşluk bırakmak zorundasınızdır.

Ayrıca `*` (yıldız) işareti Bash'te "tüm dosyalar" anlamına gelen bir joker karakter (wildcard) olduğu için, çarpma işleminde kullanırken başına ters eğik çizgi (`\`) koyarak onu "kaçırmanız" (escape etmeniz) gerekir. Yani Bash'e "bunu dosya aramak için değil, sadece çarpma sembolü olarak kullan" demiş olursunuz.

```
sonuc=$(expr $sayi1 + $sayi2)
# expr $sayi1 * $sayi2  -> HATA verir
sonuc=$(expr $sayi1 \* $sayi2) # Doğrusu
```

Sonuç: Mümkün olan her yerde `$(( ... ))` veya `(( ... ))` kullanın. Hem daha hızlı hem de daha okunabilirdir.

5.2. Temel Matematiksel Operatörler

yapısı içinde standart matematiksel operatörlerin çoğunu kullanabilirsiniz.

- `+` : Toplama
- `-` : Çıkarma
- `*` : Çarpma
- `/` : Bölme (Dikkat: Sadece tam sayı kısmını verir)
- `%` : Mod Alma (Kalanı bulma)
- `**` : Üs Alma (Kuvvet)

Örnekler:

```
a=10
b=3

echo "Toplama: $((a + b))"       # 13
echo "Çıkarma: $((a - b))"       # 7
echo "Çarpma:  $((a * b))"       # 30
echo "Bölme:   $((a / b))"       # 3 (10/3 = 3.33 ama Bash tam sayı alır)
echo "Mod:     $((a % b))"       # 1 (10'un 3'e bölümünden kalan)
echo "Üs Alma: $((a ** b))"      # 1000 (10 üzeri 3)
```

Artırma ve Azaltma Operatörleri: Döngülerde sıkça kullanacağımız C-stili operatörler de geçerlidir:

```
sayi=5
((sayi++))  # Sayıyı 1 artır (6 olur)
((sayi--))  # Sayıyı 1 azalt (5 olur)
((sayi+=10)) # Sayıya 10 ekle (15 olur)
```

5.3. Kayan Noktalı Sayılarla Çalışmak (bc kullanımı)

Bash, doğası gereği **sadece tam sayılarla (integers)** çalışır. Ondalıklı sayılar, yani programlama dünyasındaki adıyla **float** (floating point numbers) veri tipi Bash'in matematiğinde doğrudan desteklenmez. Eğer `10 / 3` işlemini Bash'e yaptırırsanız `3.33` değil, sadece `3` sonucunu alırsınız.

Ondalıklı (virgüllü/float) sayılarla işlem yapmak için harici bir hesap makinesine ihtiyacımız vardır. En yaygın kullanılan araç **`bc` (Basic Calculator)** programıdır.

Kullanımı: Matematiksel ifadenizi `echo` ile `bc`'ye "pipe" (`|`) yaparak göndeririz.

Temel Kural ve `scale`: `bc` varsayılan olarak `scale=0` ayarıyla çalışır, yani bölme işlemlerinde ondalık kısmı atar. Hassas sonuçlar almak için `scale` değerini (virgülden sonra kaç hane istediğinizi) belirtmeniz gerekir.

Örneğin, `scale=2` derseniz, sonuçlar virgülden sonra 2 basamaklı (örn: 3.33) hesaplanır.

```
echo "10 / 3" | bc
```

```
# Çıktı: 3 (Hala tam sayı)

echo "scale=2; 10 / 3" | bc
# Çıktı: 3.33
```

Değişkenlerle Kullanım:

```
a=10.5
b=2.5

# Bash bunu yapamaz: (( sonuc = a + b )) -> HATA!

# bc ile yapalım:
sonuc=$(echo "$a + $b" | bc)
echo "Sonuç: $sonuc" # Çıktı: 13.0
```

Karşılaştırmalarda bc Kullanımı: Bazen ondalıklı sayıları karşılaştırmanız da gerekebilir. `bc`, doğruysa `1`, yanlışsa `0` sonucu döner.

```
# 10.5 > 2.5 mi? (Evet -> 1)
echo "$a > $b" | bc
# Çıktı: 1

# 2.5 > 10.5 mi? (Hayır -> 0)
echo "$b > $a" | bc
# Çıktı: 0
```

Bu sonuçları ileride göreceğimiz **if** yapılarında kullanarak mantıksal kararlar alabileceğiz.

Chapter 6. Metin (String) İşlemleri

6.1. String Birleştirme (Concatenation)

Bash'te iki veya daha fazla metni birleştirmek (concatenation) diğer dillerdeki gibi `+` işaretiyle yapılmaz. Bash'te metinleri **yan yana yazmanız** yeterlidir.

Temel Kullanım:

```
isim="Ender"
soyisim="Kuş"

# Yan yana yazarak birleştirme
tam_isim="$isim $soyisim"

echo $tam_isim
# Çıktı: Ender Kuş
```

Araya Karakter Ekleyerek Birleştirme: Değişkenlerin arasına tire, nokta veya başka stringler koyabilirsiniz.

```
dosya_adi="rapor"
uzanti="txt"
tarih="2025-12-29"

tam_dosya_adi="${dosya_adi}-${tarih}.${uzanti}"

echo $tam_dosya_adi
# Çıktı: rapor-2025-12-29.txt
```

Önemli İpucu: Değişkenleri birleştirirken her zaman çift tırnak `""` ve süslü parantez `${}` kullanmak, boşluk içeren dosya isimlerinde veya karışık stringlerde hata yapmanızı önler.

6.2. String Uzunluğunu Alma

Bir metnin kaç karakterden oluştuğunu (boşluklar dahil) öğrenmek için `${#degisken}` yapısı kullanılır.

Bu yapı, aslında değişkenin "içeriğinin sayısını" verir. Dizilerde (`${#dizi[@]}`) eleman sayısını veriyordu, stringlerde ise karakter sayısını verir.

Örnek:

```
parola="Gizli123"

echo "Parola uzunluğu: ${#parola}"
```

```
# Çıktı: Parola uzunluğu: 8
```

Kullanım Alanı: Genellikle parola geçerliliği kontrol ederken (örneğin "en az 8 karakter olmalı") veya metin hizalama işlemlerinde kullanılır.

6.3. Alt Metin (Substring) Alma

Bir metnin içinden belirli bir parçayı kesip almak için dizilerde kullandığımız yöntemin aynısını kullanırız: `${degisken:başlangıç:uzunluk}`.

Temel Sözdizimi: * **Başlangıç (Offset):** Kaçıncı karakterden başlanacak? (0'dan başlar) * **Uzunluk (Length):** Kaç karakter alınacak? (Belirtilmezse sona kadar gider)

Örnek:

```
metin="LinuxDünyası"

# 0. karakterden başla, 5 karakter al
echo ${metin:0:5}
# Çıktı: Linux

# 5. karakterden başla, sona kadar git
echo ${metin:5}
# Çıktı: Dünyası

# Sondan Veri Alma (Negatif İndeks)
# Sondan 3 karakteri almak için (boşluk bırakmaya dikkat!)
echo ${metin: -3}
# Çıktı:ası
```

6.4. Metin Arama ve Değiştirme

Bir string içindeki belirli kelimeleri veya harfleri değiştirmek için `${degisken/eski/yeni}` yapısı kullanılır.

Kurallar: 1. `${degisken/aranan/yeni}` → Sadece **ilk** bulduğunu değiştirir. 2. `${degisken//aranan/yeni}` → Bulduğu **tüm** eşleşmeleri değiştirir (Global replace). 3. `${degisken/aranan}` → Aranan ifadeyi **siler** (Yeni değer boş olduğu için).

Örnek:

```
cumle="Kedi çok tatlı bir hayvandır. Kedi mırlar."

# İlk 'Kedi'yi 'Köpek' yap
echo ${cumle/Kedi/Köpek}
# Çıktı: Köpek çok tatlı bir hayvandır. Kedi mırlar.
```

```
# Tüm 'Kedi'leri 'Köpek' yap
echo ${cumle//Kedi/Köpek}
# Çıktı: Köpek çok tatlı bir hayvandır. Köpek mırlar.

# 'tatlı ' kelimesini sil
echo ${cumle/tatlı }
# Çıktı: Kedi çok bir hayvandır. Kedi mırlar.
```

6.5. Büyük/Küçük Harf Dönüşümleri

Bash 4.0 ve sonrasında gelen çok pratik bir özellik sayesinde harfleri büyütebilir veya küçültebilirsiniz.

- `^`: Sadece baş harfi büyütür.
- `^^`: Tüm harfleri büyütür.
- `,`: Sadece baş harfi küçültür.
- `,,`: Tüm harfleri küçültür.

Örnek:

```
sehir="istanbul"
kod="TR-ANT"

echo ${sehir^}   # İstanbul (Baş harf büyük)
echo ${sehir^^}  # İSTANBUL (Hepsi büyük)

echo ${kod,}     # tR-ANT (Baş harf küçük)
echo ${kod,,}    # tr-ant (Hepsi küçük)
```

Bu özellik, özellikle kullanıcı girdilerini standart hale getirmek (case-insensitive karşılaştırma yapmak) için çok kullanışlıdır.

Chapter 7. Karar Yapıları (Decision Making)

7.1. `if`, `else`, `elif` Yapıları

Bash'in karar verme mekanizması `if` (eğer) bloğudur. Bir koşulu kontrol eder ve bu koşul **doğruysa (true)** bloğun içindeki komutları çalıştırır.

Temel Sözdizimi:

```
if [ KOŞUL ]; then
    # Koşul doğruysa burası çalışır
fi
```

Not: Köşeli parantezlerin `[ ]` içinde ve etrafında BOŞLUK bırakmak zorunludur!

`else` **(Değilse) Kullanımı:**

```
if [ KOŞUL ]; then
    # Doğruysa
else
    # Yanlışsa
fi
```

`elif` **(Else If - Değilse Eğer) Kullanımı:** Birden fazla koşulu sırayla kontrol etmek için kullanılır.

```
yas=18

if [ $yas -lt 18 ]; then
    echo "Çocuksunuz."
elif [ $yas -eq 18 ]; then
    echo "Tam sınırda, 18 yaşındasınız."
else
    echo "Yetişkinsiniz."
fi
```

Bu yapı, script'lerinizin sadece düz bir çizgi gibi değil, duruma göre farklı yollara sapan akıllı programlar olmasını sağlar.

7.2. Test Operatörleri

`if` bloklarında neyi kontrol edeceğiz? İşte burada **Test Operatörleri** devreye girer. Üç ana kategoride incelenir:

1. Sayısal Karşılaştırma: * `-eq` : Eşittir (Equal) * `-ne` : Eşit değildir (Not Equal) * `-gt` : Büyüktür (Greater Than) * `-lt` : Küçüktür (Less Than) * `-ge` : Büyük eşittir (Greater or Equal) * `-le` : Küçük

eşittir (Less or Equal)

```
if [ $sayi -gt 10 ]; then ... fi
```

2. Metin (String) Karşılaştırma: * `==` : Eşittir * `!=` : Eşit değildir * `-z` : String BOŞTUR (Zero length) → Çok sık kullanılır! * `-n` : String DOLUDUR (Non-zero length)

```
ad="Ender"
if [ "$ad" == "Ender" ]; then ... fi

# Kullanıcı ad girmemiş mi kontrolü:
if [ -z "$isim" ]; then
    echo "Lütfen bir isim girin!"
fi
```

3. Dosya Kontrol Operatörleri: En güçlü kısımdır. Dosyalar ve dizinler hakkında bilgi verir. * `-e` : Dosya veya dizin VAR MI? (Exist) * `-d` : Bir DİZİN mi? (Directory) * `-f` : Normal bir DOSYA mı? (File) * `-r` : Okunabilir mi? (Readable) * `-w` : Yazılabilir mi? (Writable) * `-x` : Çalıştırılabilir mi? (Executable)

```
dosya="rapor.txt"
if [ -f "$dosya" ]; then
    echo "Dosya mevcut, üzerine yazılıyor..."
else
    echo "Hata: Dosya bulunamadı!"
fi
```

7.3. Mantıksal Operatörler (AND, OR, NOT)

Birden fazla koşulu aynı anda kontrol etmek istediğimizde mantıksal operatörler devreye girer.

1. AND (VE) Operatörü (`-a` veya `&&`): Her iki koşul da doğru olmalıdır. Not: `[[ ⋯ ]]` yapısı içinde `&&` kullanmak daha moderndir.

```
yas=25
ehliyet="var"

if [[ $yas -ge 18 && "$ehliyet" == "var" ]]; then
    echo "Araba kiralayabilirsiniz."
else
    echo "Üzgünüm, şartları sağlamıyorsunuz."
fi
```

2. OR (VEYA) Operatörü (`-o` veya `||`): İki koşuldan sadece birinin doğru olması yeterlidir.

```
gun="Cumartesi"

if [[ "$gun" == "Cumartesi" || "$gun" == "Pazar" ]]; then
    echo "Haftasonu tatili!"
fi
```

3. NOT (DEĞİL) Operatörü (`!`): Bir koşulun tersini alır. Doğruysa yanlış, yanlışsa doğru yapar.

```
if [ ! -f "config.ini" ]; then
    echo "Config dosyası EKSİK! Oluşturuluyor..."
    touch config.ini
fi
```

7.4. `case` Yapısı ile Çoklu Seçim

Eğer bir değişkenin alabileceği değerlere göre çok fazla `elif` bloğu yazmanız gerekiyorsa, `case` yapısı çok daha temiz bir alternatiftir.

Genellikle menü seçimlerinde veya komut satırı argümanlarını (ör: start, stop, restart) işlerken kullanılır.

Sözdizimi:

```
read -p "Bir meyve seçin (elma/armut/muz): " meyve

case $meyve in
    "elma")
        echo "Elma kırmızıdır."
        ;;
    "armut")
        echo "Armut sarıdır."
        ;;
    "muz")
        echo "Muz uzundur."
        ;;
    *)
        echo "Bilinmeyen meyve: $meyve"
        ;;
esac
```

Önemli Detaylar: 1. Her seçenekten sonra parantez `)` konur. 2. Yapılacak işlemlerden sonra çift noktalı virgül `;;` ile o bloğun bittiği belirtilir. 3. `*)` (yıldız), diğer hiçbir şeye uymayan durumlar için "varsayılan" (default) seçenektir (else gibi davranır). 4. `esac` (case'in tersi) ile yapı kapatılır.

7.5. Karmaşık Senaryolar ve Pekiştirme Örnekleri

Karar yapıları (Decision Making) konusunu iyice pekiştirmek için, basitten karmaşığa doğru giden gerçekçi örnekler yapalım.

Örnek 1: Basit Şifre Kontrolü (String Karşılaştırma) Kullanıcıdan gizli bir şifre isteyelim ve doğru olup olmadığına bakalım. Bu örnekte `-s` parametresini ve string eşitliğini hatırlayalım.

```
read -s -p "Lütfen parolayı girin: " parola
echo "" # Alt satıra geçmek için

if [ "$parola" == "SuperSecret123" ]; then
    echo "Giriş başarılı! Hoş geldin."
else
    echo "Hatalı parola! Erişim reddedildi."
fi
```

Örnek 2: Ondalıklı Sayı Karşılaştırma (`bc` ile) Bölüm 5'te bahsettiğimiz `bc` komutunu artık `if` içinde kullanabiliriz. Bash normalde `10.5` gibi sayıları anlamaz, ama `bc` sayesinde bunları kıyaslayabiliriz.

```
a=10.5
b=5.3

# bc -l komutu, karşılaştırma doğruysa 1, yanlışsa 0 döner.
if (( $(echo "$a > $b" | bc -l) )); then
    echo "$a sayısı $b sayısından BÜYÜKTÜR."
else
    echo "$a sayısı $b sayısından KÜÇÜKTÜR veya EŞİTTİR."
fi
```

Örnek 3: Yedekleme Aracı (Dosya Kontrolleri) Kullanıcıdan bir dosya adı alacağız. Varsa yedekleyeceğiz, yoksa uyaracağız.

```
#!/bin/bash

dosya=${1:-varsayilan.txt}

if [ -f "$dosya" ]; then
    echo "Dosya bulundu: $dosya"
    # Yedekleme işlemi
    cp "$dosya" "$dosya.bak" && echo "Yedekleme başarılı!"

elif [ -d "$dosya" ]; then
    echo "Hata: '$dosya' bir klasör! Sadece dosyaları yedekleyebilirim."

else
```

```
    echo "Hata: '$dosya' bulunamadı!"
fi
```

Örnek 4: Gelişmiş Hesap Makinesi Tüm öğrendiklerimizi (case, regex kontrolü, aritmetik) birleştiren kapsamlı bir senaryo.

```
#!/bin/bash

echo "Hesap Makinesine Hoş Geldiniz"
read -p "1. Sayı: " s1
read -p "İşlem (+, -, *, /): " islem
read -p "2. Sayı: " s2

# Basit bir regex ile sayı kontrolü
if [[ ! "$s1" =~ ^[0-9]+$ || ! "$s2" =~ ^[0-9]+$ ]]; then
    echo "Hata: Lütfen geçerli tam sayılar girin."
    exit 1
fi

case $islem in
    +)  echo "Sonuç: $((s1 + s2))" ;;
    -)  echo "Sonuç: $((s1 - s2))" ;;
    *)  # Çarpma (*) işaretini escape ediyoruz
        echo "Sonuç: $((s1 * s2))" ;;
    /)
        if [ $s2 -eq 0 ]; then
            echo "Hata: Sıfıra bölme!"
        else
            # bc ile ondalıklı bölme
            echo "Sonuç: $(echo "scale=2; $s1 / $s2" | bc)"
        fi
        ;;
    *)  echo "Geçersiz işlem!" ;;
esac
```

Chapter 8. Döngüler

8.1. `for` Döngüsü

Döngü Nedir? Döngüler, belirli bir işlemi tekrar tekrar yapmamızı sağlayan yapılardır. "Bu komutu 100 kere çalıştır" veya "Şu klasördeki tüm dosyalar için sırayla şu işlemi yap" gibi görevler için döngüleri kullanırız. Elle tek tek yazmak yerine, döngüye bir kere tarif ederiz, o bizim yerimize defalarca çalıştırır.

Bash'te `for` döngüsünün iki temel kullanım şekli vardır.

1. Liste Üzerinde Gezinme (Standart Bash Tarzı): Bash'in en doğal ve sık kullanılan döngü türüdür. Bir kelime listesinin, dosya grubunun veya komut çıktısının üzerinde tek tek gezinmek için kullanılır.

```
# Basit Liste: 'isim' değişkeni sırayla Ali, Ayşe ve Ahmet değerlerini alır.
for isim in Ali Ayşe Ahmet; do
    echo "Merhaba $isim"
done
```

Aralık Belirtme: `{başlangıç..bitiş}` yapısı ile sayı aralığı verebilirsiniz.

```
for sayi in {1..5}; do
    echo "Sayı: $sayi"
done
```

2. C-Tarzı `for` Döngüsü: Eğer C, Java, PHP veya JavaScript gibi dillerden geliyorsanız bu yapı size çok tanıdık gelecektir. Genellikle matematiksel sayaçlar için kullanılır.

Bu yapıda **Çift Parantez** `(( ... ))` kullanılır ve 3 bölümden oluşur: 1. **Başlangıç (Initialization):** Döngü hangi sayıdan başlayacak? (`i=1`) 2. **Koşul (Condition):** Döngü ne zamana kadar dönecek? (`i<=5`, yani i 5'ten küçük veya eşit olduğu sürece) 3. **Artış (Increment):** Her adımda sayaç nasıl değişecek? (`i++`, yani i'yi 1 artır)

```
# i=1'den başla; i 5'e eşit veya küçükse devam et; her turda i'yi 1 artır.
for (( i=1; i<=5; i++ )); do
    echo "Sayaç: $i"
done
```

Bu yapı, özellikle "Listenin 3. elemanından başla, ikişer ikişer git" gibi karmaşık matematiksel döngüler kurarken `for i in {1..100..2}` yapısına göre daha esnek olabilir.

8.2. `while` ve `until` Döngüleri

Koşula bağlı döngülerdir.

1. `while` Döngüsü: Koşul **doğru olduğu sürece** döner. (While = -iken, süresince)

```
sayac=1

while [ $sayac -le 5 ]; do
    echo "Sayac: $sayac"
    ((sayac++)) # Sayacı artırmayı unutmayın, yoksa sonsuz döngü olur!
done
```

Sonsuz Döngü: Genellikle servisleri veya sürekli çalışan scriptleri yazarken kullanılır.

```
while true; do
    echo "Çalışıyorum..."
    sleep 1 # 1 saniye bekle
done
```

2. `until` Döngüsü: Koşul **doğru OLANA KADAR** döner. Yani koşul yanlışken çalışır, doğru olunca durur. (Until = -e kadar) `while'ın tam tersidir.

```
sayi=10

# Sayı 0'a eşit OLANA KADAR dön
until [ $sayi -eq 0 ]; do
    echo "Geri sayım: $sayi"
    ((sayi--))
done
```

8.3. Döngü Kontrolü: `break` ve `continue`

Bazen döngü bitmeden çıkmak veya o anki adımı atlamak isteriz.

1. `break`: Döngüyü Kırıp Çık Aradığımız bir şeyi bulduğumuzda döngünün geri kalanını çalıştırmaya gerek yoktur.

```
for sayi in {1..10}; do
    if [ $sayi -eq 5 ]; then
        echo "5 bulundu! Çıkılıyor..."
        break
    fi
    echo "Sayı: $sayi"
done
```

Çıktı sadece 1'den 4'e kadar sayıları basar, 5'i bulunca "5 bulundu!" der ve çıkar.

2. `continue`: Sadece Bu Adımı Atla Döngüyü kırmaz, sadece o anki iterasyonu (turu) pas geçer ve

bir sonraki tura geçer.

```
for sayi in {1..5}; do
    if [ $sayi -eq 3 ]; then
        echo "3'ü atlıyorum..."
        continue
    fi
    echo "İşlenen sayı: $sayi"
done
```

Çıktıda 3 hariç diğer tüm sayılar (1, 2, 4, 5) görünür.

8.4. Dosya Satırlarını Okuma Döngüsü

Bir dosyanın içeriğini satır satır okuyup işlem yapmak, sistem yönetiminde en sık yapılan işlerden biridir. Bunu `while` döngüsü ve `read` komutu ile yaparız.

Temel Kalıp:

```
while read -r satir; do
    echo "Okunan satır: $satir"
done < "dosya.txt"
```

Neden `-r`? `read` komutuna `-r` parametresini vermek, ters eğik çizgi (`\`) karakterinin kaçış karakteri olarak yorumlanmasını engeller. Dosyadaki veriyi olduğu gibi okumak için her zaman `-r` kullanmak en güvenli yoldur.

Örnek Uygulama: `sunucular.txt` adında bir dosyamız olsun ve içinde sunucu IP'leri bulunsun. Her birine ping atmak istiyoruz:

```
dosya="sunucular.txt"

# Dosya var mı kontrol edelim
if [ ! -f "$dosya" ]; then
    echo "Hata: $dosya bulunamadı."
    exit 1
fi

while read -r ip; do
    echo "Ping atılıyor: $ip"
    ping -c 1 "$ip" &> /dev/null # Çıktıyı gizle

    if [ $? -eq 0 ]; then
        echo "  -> $ip ERIŞILEBILIR."
    else
        echo "  -> $ip ERIŞILEMEZ!"
    fi
```

```
done < "$dosya"
```

Bu yapı, yüzlerce sunucuyu tek komutla kontrol etmenizi sağlar.

8.5. Döngüler İçin Pekiştirme Örnekleri

Döngüler konusunu daha iyi kavramak için günlük hayatta işinize yarayacak pratik senaryolar hazırladık.

Senaryo 1: Toplu Dosya Uzantısı Değiştirme Elimizde `.htm` uzantılı birçok dosya var ve bunları `.html` yapmak istiyoruz.

```
#!/bin/bash

echo "Dönüştürme başlıyor..."

for dosya in *.htm; do
    # Eğer hiç .htm dosyası yoksa döngü *.htm diye ham string döner, bunu engellemek
için:
    [ -e "$dosya" ] || continue

    # ${degisken%desen} yapısı ile sondaki .htm'i atıp yeni isim oluşturuyoruz
    yeni_isim="${dosya%.htm}.html"

    mv "$dosya" "$yeni_isim"
    echo "$dosya -> $yeni_isim olarak değiştirildi."
done
```

Senaryo 2: Geri Sayım Aracı (Until Kullanımı) Kullanıcıdan bir saniye değeri alıp geri sayan bir araç.

```
#!/bin/bash

read -p "Kaç saniye geri sayayım? " saniye

until [ $saniye -lt 1 ]; do
    echo "$saniye..."
    sleep 1
    ((saniye--))
done

echo "Süre doldu! "
```

Senaryo 3: Kullanıcı Oluşturma (Dosyadan Okuma) `kullanicilar.txt` dosyasındaki isimleri okuyup sisteme ekleyen (simülasyon) bir script.

```
while read -r kullanici; do
```

```
    # Boş satırları atla
    if [ -z "$kullanici" ]; then continue; fi

    echo "Kullanıcı oluşturuluyor: $kullanici"
    # sudo useradd -m "$kullanici" (Gerçek kullanım için bu satırı açın)
    echo "  -> $kullanici eklendi."

done < "kullanicilar.txt"
```

Bu örnekler, döngülerin sadece sayı saymak için değil, dosya sistemi ve sistem yönetimi için ne kadar güçlü olduğunu göstermektedir.

Chapter 9. Fonksiyonlar

9.1. Fonksiyon Tanımlama ve Çağırma

Yazılım dünyasında altın bir kural vardır: **DRY (Don't Repeat Yourself)**, yani "Kendini Tekrar Etme". Eğer bir script içinde aynı kodu iki veya daha fazla kez yazıyorsanız, onu bir **fonksiyon** haline getirmelisiniz.

Fonksiyonlar, karmaşık scriptleri küçük, yönetilebilir ve anlaşılır parçalara böler.

Tanımlama Yöntemleri: Bash'te fonksiyon tanımlamanın iki yolu vardır. İkisi de aynı işi yapar, ancak yazım farkı vardır.

Yöntem 1: Standart (POSIX) ve Önerilen En yaygın ve taşınabilir yöntem budur. `function` kelimesi kullanılmaz, sadece parantez `()` yeterlidir.

```
selamla() {
    echo "Merhaba Dünya!"
}
```

Yöntem 2: `function` Anahtar Kelimesi ile Bazı programcılar daha okunaklı bulduğu için bunu kullanır.

```
function selamla {
    echo "Merhaba Dünya!"
}
```

Fonksiyon İsimlendirme: Fonksiyon isimleri açıklayıcı olmalıdır. Genellikle küçük harf ve kelime aralarında alt çizgi (`snake_case`) kullanılır (örn: `dosya_kontrol_et`, `log_yaz`).

Çağırma (Çalıştırma): Fonksiyonu tanımladığınızda kod hemen çalışmaz; sadece hafızaya alınır. Çalıştırmak için script'in ana akışında ismini yazmalısınız.

Önemli Kurallar: 1. **Parantez Yok:** Çağırırken `selamla()` değil, sadece `selamla` yazılır. 2. **Sıralama:** Bash scriptleri yukarıdan aşağıya okunur. Fonksiyonu **çağırmadan ÖNCE** tanımlamalısınız.

```
#!/bin/bash

# HATA! Bash henüz 'baslat' diye bir şey bilmiyor.
# baslat

baslat() {
    echo "Sistem başlatılıyor..."
}

# DOĞRU: Tanımlandıktan sonra çağrılır.
```

```
baslat
```

9.2. Fonksiyonlara Parametre Gönderme

Diğer programlama dillerinde fonksiyon parametreleri parantez içinde tanımlanır (`func(a, b)` gibi). **Bash'te ise durum farklıdır.**

Bash fonksiyonlarına parametreler, tıpkı script'e argüman gönderir gibi gönderilir. Fonksiyon içinde bu parametrelere `$1`, `$2` gibi özel değişkenlerle erişilir.

Örnek:

```
#!/bin/bash

# Fonksiyon Tanımı
topla() {
    sayi1=$1
    sayi2=$2

    echo "Toplam: $((sayi1 + sayi2))"
}

# Fonksiyonu Çağırma (Parametreleri yanına yazıyoruz)
topla 10 20
topla 5 8
```

Dikkat: Fonksiyon içindeki `$1`, script'e gönderilen `$1` (ana argüman) ile karışmaz. Fonksiyon çalıştığı süre boyunca `$1`, fonksiyona gönderilen ilk parametredir.

9.3. Return Değeri ve Exit Status

Bash'e yeni başlayanların en çok karıştırdığı nokta "Return" (Dönüş) kavramıdır. Diğer dillerde (Python, Java vb.) `return "Sonuç"` diyerek metin veya nesne döndürebilirsiniz. **Bash'te bu MÜMKÜN DEĞİLDİR.**

Bash'te bir fonksiyonun dış dünyaya veri iletmesinin iki yolu vardır:

1. **Exit Status (Çıkış Kodu):** Başarı `0` veya hata durumunu `1-255` bildirmek için. (`return` komutu ile)
2. **Standart Çıktı (Stdout):** Gerçek veriyi (metin, sayı) göndermek için. (`echo` komutu ile)

1. `return` ile Durum Bildirme: `return` komutu sadece **0 ile 255** arasında bir tam sayı döndürebilir. Bu, "Fonksiyon görevini başarıyla tamamladı mı?" sorusunun cevabıdır.

- `return 0`: Başarılı (True gibi düşünün)
- `return 1` (veya üzeri): Hata (False gibi düşünün)

```
dosya_kontrol() {
    if [ -f "$1" ]; then
        return 0 # Başarılı
    else
        return 1 # Hata
    fi
}

dosya_kontrol "rapor.txt"

# $? değişkeni son çalışan komutun/fonksiyonun çıkış kodunu verir.
if [ $? -eq 0 ]; then
    echo "Dosya mevcut."
else
    echo "Dosya yok!"
fi
```

2. String Döndürmek için `echo` Trick'i: Eğer fonksiyonun bir veri (string, hesaplama sonucu vb.) üretmesini istiyorsanız, bunu `echo` ile basıp, fonksiyonu çağırırken Command Substitution `$(…)` kullanmalısınız.

```
buyuk_harf_yap() {
    local metin="$1"
    echo "${metin^^}"
}

# Fonksiyonun çıktısını değişkene atıyoruz
sonuc=$(buyuk_harf_yap "merhaba")
echo "Dönülen değer: $sonuc"
```

Bu yöntem Bash scriptlerinde "değer döndürme"nin (return value) standart yoludur.

9.4. Değişken Kapsamı (Local vs Global)

Bash'te varsayılan olarak bir fonksiyon içinde tanımladığınız değişken **GLOBAL** olur. Yani fonksiyon bittikten sonra bile o değişkene dışarıdan erişilebilir ve bu durum istenmeyen hatalara yol açabilir.

Bunu engellemek için `local` anahtar kelimesi kullanılır.

Hatalı Kullanım (Global):

```
isim="Ender"

degistir() {
    isim="Ali" # Global olan 'isim' değişkenini ezer!
}
```

```
degistir
echo $isim
# Çıktı: Ali (Orijinal değer bozuldu!)
```

Doğru Kullanım (Local):

```
isim="Ender"

degistir() {
    local isim="Ali" # Sadece bu fonksiyon içinde geçerli yeni bir değişken
    echo "Fonksiyon içi: $isim"
}

degistir
echo "Fonksiyon dışı: $isim"
# Çıktı:
# Fonksiyon içi: Ali
# Fonksiyon dışı: Ender (Korundu!)
```

Kural: Fonksiyon içinde tanımladığınız her değişkenin başına mutlaka `local` koyun.

9.5. Kütüphane Oluşturma ve `source` Kullanımı

Büyük projelerde tüm fonksiyonları tek bir dosyaya yazmak yönetimi zorlaştırır. Bunun yerine fonksiyonları ayrı dosyalara (kütüphanelere) yazıp ana script'ten çağırabiliriz.

Örneğin `utils.sh` adında bir dosyamız olsun:

```
# utils.sh dosyasi
log_yaz() {
    echo "[LOG] $1"
}
```

Ana scriptimizden (`main.sh`) bunu kullanmak için `source` veya kısaca `.` komutunu kullanırız.

```
#!/bin/bash
# main.sh

source ./utils.sh
# veya
# . ./utils.sh

log_yaz "Script başladı."
```

Bu işlem, `utils.sh` dosyasının içeriğini sanki o noktaya kopyalayıp yapıştırmışsınız gibi çalıştırır ve

içindeki fonksiyonları erişilebilir kılar.

9.6. Kapsamlı Fonksiyon Örnekleri

Fonksiyonlar konusunu pekiştirmek için, önceki bölümlerde öğrendiğimiz karar yapıları, döngüler ve aritmetik işlemleri de içeren gerçekçi senaryoları inceleyelim.

Örnek 1: Log Tutma Fonksiyonu (Renkli ve Tarihli) Her script'te ihtiyaç duyulan, ekrana tarih ve saat ile birlikte renkli hata/bilgi mesajı basan bir fonksiyon.

```
#!/bin/bash

# Renk kodları
KIRMIZI='\033[0;31m'
YESIL='\033[0;32m'
NORMAL='\033[0m'

log_yaz() {
    local tur="$1"
    local mesaj="$2"
    local tarih=$(date "+%Y-%m-%d %H:%M:%S")

    case $tur in
        "BILGI")
            echo -e "${YESIL}[$tarih] [BILGI] $mesaj${NORMAL}"
            ;;
        "HATA")
            echo -e "${KIRMIZI}[$tarih] [HATA] $mesaj${NORMAL}"
            ;;
        *)
            echo "[$tarih] $mesaj"
            ;;
    esac
}

# Kullanımı
log_yaz "BILGI" "Script başlatıldı."
log_yaz "HATA" "Dosya bulunamadı!"
```

Örnek 2: Matematiksel Hesaplama ve Sonuç Döndürme Kullanıcıdan alınan sayının karesini hesaplayıp, sonucu değişkene atayalım. Burada `echo` ile değer döndürme tekniğini kullanacağız.

```
#!/bin/bash

karesini_al() {
    local sayi=$1
    # Kontrol: Sayısal değilse hata dön (return 1)
    if [[ ! "$sayi" =~ ^[0-9]+$ ]]; then
```

```
        return 1
    fi

    # Hesaplamayı yap ve echo ile bas (Bu bizim return değerimiz olacak)
    local kare=$((sayi * sayi))
    echo $kare
}

read -p "Bir sayı girin: " girilen

# Fonksiyonu $(...) içinde çağırıyoruz ki echo çıktısını yakalayalım
sonuc=$(karesini_al "$girilen")

# $? ile fonksiyonun return kodunu (exit status) kontrol edelim
if [ $? -eq 0 ]; then
    echo "Sayının karesi: $sonuc"
else
    echo "Hata: Lütfen geçerli bir sayı giriniz!"
fi
```

Örnek 3: Servis Kontrol Fonksiyonu Parametre olarak gönderilen bir servisin (örn: nginx, docker) çalışıp çalışmadığını kontrol eden ve duruma göre işlem yapan fonksiyon.

```
#!/bin/bash

servis_kontrol() {
    local servis_adi=$1

    # systemctl komutunun çıktısını sessize alıyoruz (> /dev/null)
    # is-active komutu çalışıyorsa 0, çalışmıyorsa farklı kod döner
    if systemctl is-active --quiet "$servis_adi"; then
        return 0
    else
        return 1
    fi
}

hedef_servis="nginx"

# if bloğunda doğrudan fonksiyon çağrılabilir
# Çünkü if, fonksiyonun return değerine (0 mı 1 mi) bakar.
if servis_kontrol "$hedef_servis"; then
    echo "$hedef_servis çalışıyor, her şey yolunda."
else
    echo "UYARI: $hedef_servis çalışmıyor! Başlatılıyor..."
    # sudo systemctl start "$hedef_servis"
fi
```

Chapter 10. Girdi/Çıktı Yönlendirme ve Dosya İşlemleri

10.1. Standart Akışlar (Stdin, Stdout, Stderr)

Linux dünyasında "Her şey bir dosyadır" felsefesi hakimdir. Çalıştırdığınız her komut veya program, işletim sistemiyle iletişim kurmak için arka planda **3 tane veri kanalı (akış)** açar. Bu kanalları anlamak, Bash scriptçiliğinin en önemli adımıdır.

Bu üç kanalın her birinin bir numarası (File Descriptor - FD) vardır:

1. **Stdin (Standard Input) - [No: 0]:**
    - **Giriş Kanalı:** Programın veri aldığı yerdir.
    - Varsayılan olarak **Klavyemizdir**.
2. **Stdout (Standard Output) - [No: 1]:**
    - **Çıkış Kanalı:** Programın ürettiği başarılı sonuçların gittiği yerdir.
    - Varsayılan olarak **Terminal Ekranıdır**.
3. **Stderr (Standard Error) - [No: 2]:**
    - **Hata Kanalı:** Programın ürettiği hata mesajlarının gittiği yerdir.
    - Varsayılan olarak bu da **Terminal Ekranıdır**.

Neden Ayrılmışlar? Hata mesajları ile gerçek sonuçların birbirine karışmaması için ayrılmışlardır. Örneğin bir yedekleme scriptinin çıktısını dosyaya kaydederken, hataları ekranda görmek isteyebilirsiniz.

10.2. Yönlendirme Operatörleri (>, >>, 2>)

Standart akışların yönünü değiştirmek için "Yönlendirme Operatörleri"ni kullanırız. Yani çıktıyı ekrana değil dosyaya yazabilir veya girdiyi klavyeden değil dosyadan okuyabiliriz.

1. Çıktı Yönlendirme (> ve >>): Stdout (1) akışını dosyaya yönlendirir.

- `>` **(Oluştur/Ez):** Dosya yoksa oluşturur, varsa **içini silip** baştan yazar.
- `>>` **(Ekle - Append):** Dosya yoksa oluşturur, varsa **sonuna ekler**. Eski veriler silinmez.

```
echo "Bu satır dosyayı ezer." > dosya.txt
echo "Bu satır sonuna eklenir." >> dosya.txt
```

2. Hata Yönlendirme (2>): Sadece Stderr (2) akışını, yani hata mesajlarını yakalar.

```
# Olmayan bir dosyayı listelemeye çalışalım (Hata verir)
ls olmayan_dosya 2> hatalar.log
```

```
# Ekranda hiçbir şey görmezsiniz, hata mesajı 'hatalar.log'a yazılır.
```

3. Hepsini Yönlendirme (`&>` veya `2>&1`): Hem başarılı çıktıları hem de hataları aynı dosyaya göndermek için kullanılır.

```
# Modern Bash kullanımı (Önerilen)
komut &> hepsi.log

# Eski yöntem (Stdout dosyaya, Stderr ise Stdout'a)
komut > hepsi.log 2>&1
```

4. Özel Dosya: `/dev/null` Linux'un kara deliğidir. Buraya gönderilen her şey yok olur. Bir komutun çıktısını veya hatasını görmek istemiyorsanız buraya atabilirsiniz.

```
# Çıktıları görmezden gel, sessizce çalış
ping -c 1 google.com > /dev/null 2>&1
```

10.3. Pipe (|) Kullanımı ve Filtreleme

Linux'un en güçlü özelliklerinden biri **Pipe (Boru)** sistemidir. Pipe, bir komutun çıktısını (stdout) alıp, doğrudan başka bir komutun girdisine (stdin) bağlar. Bu, zincirleme işlem yapmayı sağlar.

Mantık: `Komut1 | Komut2 | Komut3`

Örnek Senaryo: Çalışan süreçler (`ps aux`) arasında ismi "bash" olanları bulmak (`grep`), sonra bunları sıralamak (`sort`) ve sadece ilk 5 tanesini görmek (`head`) istiyoruz.

```
ps aux | grep "bash" | sort | head -n 5
```

Burada hiçbir ara dosya oluşturulmaz. Veri bir komuttan diğerine "akar".

Sık Kullanılan Filtre Komutları: Pipe ile birlikte kullanılan bazı temel komutlar şunlardır:

- **`grep`:** Metin içinde arama yapar/filtreler.
- **`sort`:** Satırları alfabetik veya sayısal sıralar.
- **`uniq`:** Tekrarlayan satırları eler (Sadece sıralı veride çalışır, bu yüzden önce `sort` yapılır).
- **`head` / `tail`:** Dosyanın başını veya sonunu gösterir.
- **`wc`:** Satır, kelime ve karakter sayar (`wc -l` satır sayar).

```
# Bir klasörde kaç tane .txt dosyası var?
ls *.txt | wc -l
```

10.4. Dosya Varlığı ve İzin Kontrolleri (Tekrar)

Bölüm 7'de dosya kontrol operatörlerini (`-f`, `-d`) görmüştük. Ancak dosya işlemlerinde güvenlik ve hatasız çalışma için bilinmesi gereken birkaç detay daha vardır.

1. Dosya Yazılabilir mi? (`-w`) Bir log dosyasına yazmadan önce, oraya yazma iznimiz olup olmadığını kontrol etmek iyi bir pratiktir.

```
dosya="/var/log/ozel.log"

if [ -w "$dosya" ]; then
    echo "Log yazılıyor..." >> "$dosya"
else
    echo "Hata: Dosyaya yazma izniniz yok!" >&2
    exit 1
fi
```

2. Dosya Boş mu? (`-s`) `-s` (size) operatörü, dosya varsa ve boyutu 0'dan büyükse TRUE döner. Dosyanın içinin boş olup olmadığını anlamak için kullanılır.

```
if [ -s "veri.txt" ]; then
    echo "Dosya dolu, işlem başlatılıyor."
else
    echo "Dosya boş veya yok!"
fi
```

3. `/dev/null` ile Çöp Kutusu Bazen bir komutun çıktısıyla hiç ilgilenmeyiz. Sadece hata verip vermediği önemlidir.

```
# grep çıktısı ekrana gelmesin, sadece bulup bulmadığına (exit status) bakalım
if grep -q "Hata" log.txt; then
    echo "Log dosyasında hata bulundu!"
fi
```

Not: `grep -q` (quiet), çıktıyı `/dev/null'a göndermekle eşdeğerdir ve daha hızlıdır.

Chapter 11. Otomasyon ve İleri Konular

11.1. Cron ile Zamanlanmış Görevler

Sistem yöneticiliğinin olmazsa olmazı otomasyondur. Bir scripti her gece 03:00'te çalıştırmak veya her 5 dakikada bir sunucuyu kontrol etmek için **Cron** servisini kullanırız. Cron, arka planda sessizce çalışan ve zamanı gelen işleri tetikleyen bir "zamanlayıcı daemon"dur.

Crontab (Cron Table) Nedir? Her kullanıcının kendine ait bir görev çizelgesi (tablosu) vardır. Bu tabloyu düzenlemek için terminale şu komutu yazarız: `crontab -e`

> [!NOTE] `sudo crontab -e` komutu root kullanıcısının zamanlayıcısını açar. Normal kullanıcı ile root'un görevleri ayrı dosyalarda tutulur.

Zamanlama Sözdizimi (Syntax): Cron dosyası 5 yıldızlı bir zamanlayıcı sistemi kullanır:

```
* * * * * /komut/yolu
```

1. **Dakika** (0-59)
2. **Saat** (0-23)
3. **Ayın Günü** (1-31)
4. **Ay** (1-12)
5. **Haftanın Günü** (0-6) (0=Pazar, 1=Pazartesi...)

Zamanlama İçin Özel Karakterler: * `: Her (Her dakika, her saat⋯) * , : Liste (Örn: 1,15,30 → 1., 15. ve 30. dakikalarda) * - : Aralık (Örn: 1-5 → 1'den 5'e kadar) * / : Periyot/Tekrar (Örn:/10` → Her 10 dakikada bir)

Pratik Örnekler:

```
# Her gece saat 03:00'te yedek al
0 3 * * * /home/kullanici/scripts/yedekle.sh

# Hafta içi her gün (Pzt-Cum) sabah 08:30'da rapor gönder
30 8 * * 1-5 /home/kullanici/scripts/raporla.sh

# Her 15 dakikada bir çalış
*/15 * * * * /home/kullanici/scripts/kontrol.sh
```

Özel Kısayollar (Special Strings): 5 yıldızla uğraşmak istemeyenler için pratik takma adlar vardır: * `@reboot` : Sunucu her yeniden başladığında 1 kez çalışır (Çok kullanışlıdır!). * `@daily` veya `@midnight` : Her gece 00:00'da çalışır (`0 0 * * *`). * `@hourly` : Her saat başı çalışır (`0 * * * *`).

Çıktı ve Hata Yönetimi (ÖNEMLİ): Cron, çalıştırdığı komutların çıktısını (stdout/stderr) varsayılan olarak kullanıcıya **e-posta** olarak atmaya çalışır. Sunucuda mail kurulumu yoksa bu çıktılar kaybolur veya hata verir. Bu yüzden çıktıları mutlaka log dosyasına yönlendirin:

```
# Çıktıları ve hataları loga yaz
* * * * * /home/user/script.sh >> /var/log/cron_script.log 2>&1

# Çıktıları çöpe at (Sessiz mod)
* * * * * /home/user/script.sh > /dev/null 2>&1
```

Altın Kural: PATH Sorunu Cron, sizin terminalinizdeki çevre değişkenlerini (`PATH`) bilmez. Terminalde `python main.py` yazarsınız çalışır, ama Cron'da çalışmaz. Çözüm: 1. **Tam Yol Kullanın:** `python` yerine `/usr/bin/python3`, `yedek.sh` yerine `/home/user/yedek.sh`. 2. **Script İçinde PATH Tanımlayın:** Scriptinizin en başına `PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin` ekleyin.

11.2. Hata Ayıklama (Debugging) Teknikleri

Yazdığınız script çalışmıyor veya beklenmedik sonuçlar veriyorsa, hatanın kaynağını bulmak için Bash'in sunduğu hata ayıklama modlarını kullanabilirsiniz.

1. `set -x` (X-Ray Modu): Scriptin çalışırken hangi komutları işlettiğini, değişkenlerin o anki değerlerini ekrana basar. Hatanın nerede olduğunu görmek için mükemmeldir.

```
#!/bin/bash
set -x  # Hata ayıklamayı başlat

ad="Ender"
echo "Merhaba $ad"

set +x  # Hata ayıklamayı kapat
```

2. Scripti Debug Modunda Çalıştırmak: Scriptin içine kod yazmadan, çalıştırırken de bu modu açabilirsiniz: `bash -x scriptiniz.sh`

3. `set -e` (Hata Olursa Dur): Normalde Bash, bir satırda hata olsa bile diğer satıra geçip çalışmaya devam eder (bu bazen felakete yol açabilir). `set -e` komutu, **herhangi bir komut hata verirse (exit code != 0) scripti anında durdurur.**

```
#!/bin/bash
set -e

cd /olmayan/klasor  # Burası hata verecek
rm -rf *            # set -e olduğu için bu tehlikeli komut ASLA çalışmaz!
```

Güvenli scriptler yazmak için scriptlerinizin başına `set -e` eklemeyi alışkanlık haline getirin.

11.3. Temel Regex (Düzenli İfadeler) Kullanımı

Regular Expressions (Regex), metinler içinde karmaşık desenleri tanımlamak ve aramak için kullanılan evrensel bir dildir. Bash'te özellikle `grep` komutuyla ve `[[ ⋯ =~ ⋯ ]]` koşul yapısında kullanılır.

Temel Semboller: * `^` : Satır başı (Örn: `^Hata` → Satır "Hata" ile başlıyorsa) * `$` : Satır sonu (Örn: `tamam$` → Satır "tamam" ile bitiyorsa) * `.` : Herhangi bir tek karakter * `*` : Kendinden önceki karakterden 0 veya daha fazla tekrar * `[ ]` : Karakter kümesi (Örn: `[0-9]` rakamlar, `[a-z]` küçük harfler)

Örnekler:

```
# E-posta formatı kontrolü (Basit)
read -p "Mail adresi: " mail

if [[ "$mail" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Format geçerli."
else
    echo "Geçersiz mail formatı!"
fi

# Bir dosyada sadece rakamlardan oluşan satırları bul
grep "^[0-9]*$" veriler.txt
```

Regex öğrenmek ayrı bir kitap konusudur, ancak bu temel sembolleri bilmek çoğu işinizi çözecektir.

11.4. `awk` ve `sed` ile Metin İşlemeye Giriş

Bash'in kendi string özellikleri (`${var⋯}`) basit işlemler için yeterlidir. Ancak elimizde binlerce satırlık log dosyaları, CSV verileri veya karmaşık metin yığınları varsa, sahneye Linux'un iki süper kahramanı çıkar: `sed` ve `awk`.

Bu araçlar aslında kendi başlarına birer programlama dilidir. Bash scriptlerinizde bunları "çağırıp" işleri yaptırırsınız.

1. `sed` (Stream Editor): Metin Cerrahı `sed`, bir metni satır satır okur ve üzerinde silme, değiştirme, ekleme işlemleri yapar. En büyük gücü "Bul ve Değiştir" özelliğidir.

- **Temel Değiştirme (Substitute):** `s/eski/yeni/` kalıbı kullanılır.

```
# 'elma' kelimesini 'armut' yap (Sadece ekrana basar, orijinal dosyaya dokunmaz)
sed 's/elma/armut/' dosya.txt

# 'g' (global) parametresi ile satırdaki TÜM 'elma'ları değiştirir (yoksa sadece
ilkini yapar)
```

```
sed 's/elma/armut/g' dosya.txt
```

- **Dosyayı Yerinde Değiştirme (`-i`):** Normalde `sed` sonucu alır ekrana basar. Dosyayı kalıcı değiştirmek için `-i` (in-place) kullanılır. > [!WARNING] > `-i` kullanırken dikkatli olun! Hata yaparsanız dosyanız bozulur. Güvenlik için `-i.bak` kullanarak yedeğini alabilirsiniz.

```
# dosya.txt.bak yedeğini oluşturur, sonra dosya.txt'yi değiştirir
sed -i.bak 's/eski/yeni/g' dosya.txt
```

- **Ayraç Sorunu (Slash `/` Karakteri):** Eğer `/usr/bin/python` gibi slash içeren bir metni değiştirecekseniz, `s///` yapısı karışır. Bunun yerine başka bir karakter (örn: `#` veya `|`) kullanabilirsiniz.

```
# /var/www yolunu /home/web ile değiştir
sed 's#/var/www#/home/web#g' config.conf
```

2. `awk`: Sütun ve Veri Analisti `awk` verileri satır (record) ve sütun (field) olarak görür. Özellikle boşlukla veya virgülle ayrılmış verilerde (loglar, CSV) inanılmaz güçlüdür.

- **Sütun Seçme:** `$1`, `$2`... sütunları temsil eder. `$0` satırın tamamıdır.

```
# 'ls -l' çıktısından sadece Dosya Boyutu ($5) ve Dosya Adını ($9) al
ls -l | awk '{print $5, $9}'
```

- **Özel Ayraç Belirleme (`-F`):** Varsayılan olarak boşluğa göre böler. Eğer `/etc/passwd` gibi `:` ile ayrılmış bir dosya okuyorsanız:

```
# Kullanıcı adlarını ($1) yazdır
awk -F':' '{print $1}' /etc/passwd
```

- **Dahili Değişkenler (NR, NF):**
- **NR (Number of Records):** Kaçıncı satırdayız?
- **NF (Number of Fields):** O satırda kaç sütun var?

```
# Sadece 5 sütundan fazla veriye sahip satırları yazdır
awk 'NF > 5 {print $0}' log.txt

# Satır numarası ekleyerek yazdır
awk '{print NR, $0}' liste.txt
```

Hangisini Ne Zaman Kullanmalı? * `sed`: Basit kelime değişiklikleri, satır silme/ekleme işlemleri için. (Metni **düzenler**) * `awk`: Sütun bazlı veri çekme, hesaplama yapma (toplama/ortalama), mantıksal süzme işlemleri için. (Veriyi **analiz eder**)

11.5. Loglama ve Hata Yönetimi

Profesyonel bir script, ne yaptığını kaydeder ve hataları düzgün yönetir.

Neden Log Tutmalıyız? * Hata olduğunda geriye dönüp neyin yanlış gittiğini anlamak için. * Zamanlanmış görevlerin (cron) çalışıp çalışmadığını doğrulamak için.

Basit Loglama Fonksiyonu:

```
log_dosyasi="/var/log/yedekleme.log"

logla() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> "$log_dosyasi"
}

logla "Yedekleme işlemi başladı."

# Bir komut çalıştıralım ve sonucuna göre log yazalım
if cp veriler.txt /yedek/ ; then
    logla "Kopyalama BAŞARILI."
else
    # Hatayı hem loga hem de Stderr'e yazalım
    hata_mesaji="Kopyalama BAŞARISIZ oldu!"
    logla "[HATA] $hata_mesaji"
    echo "$hata_mesaji" >&2
    exit 1
fi
```

Hata Yakalama (Trap): Script aniden sonlandırılırsa (CTRL+C) veya hata alıp kapanırsa, arkada çöp dosya (`/tmp/⋯`) bırakmamak için `trap` komutu kullanılır.

```
temizlik_yap() {
    echo "Script kapatılıyor, geçici dosyalar siliniyor..."
    rm -f /tmp/gecici_dosya.txt
}

# EXIT sinyali (her türlü çıkış) gelince 'temizlik_yap' fonksiyonunu çalıştır
trap temizlik_yap EXIT

touch /tmp/gecici_dosya.txt
# İşlemler...
# Script bittiğinde 'temizlik_yap' otomatik çalışacaktır.
```

Chapter 12. Pratik Senaryolar ve Projeler

12.1. Proje 1: Sistem Bilgisi Raporlama Aracı

Bu projede, sunucunun o anki durumunu (CPU, RAM, Disk kullanımı) analiz eden ve bize okunabilir bir rapor sunan bir script yazacağız.

Amaç: * Değişkenleri ve komut değiştirmeyi (`$()`) kullanmak. * Aritmetik işlemler yapmak. * Renkli çıktı üretmek.

Script Kodu (`sistem_raporu.sh`):

```
#!/bin/bash

# --- Ayarlar ve Renkler ---
SIYAH='\033[0;30m'
KIRMIZI='\033[0;31m'
YESIL='\033[0;32m'
SARI='\033[0;33m'
MAVI='\033[0;34m'
NORMAL='\033[0m'

baslik_yaz() {
    echo -e "\n${MAVI}=== $1 ===${NORMAL}"
}

# --- 1. Temel Bilgiler ---
baslik_yaz "SİSTEM BİLGİLERİ"
echo "Hostname    : $(hostname)"
echo "Tarih/Saat  : $(date)"
echo "Uptime      : $(uptime -p)"
echo "Çekirdek    : $(uname -r)"

# --- 2. Disk Kullanımı ---
baslik_yaz "DİSK DURUMU (Kök Dizin)"
disk_kullanim=$(df -h / | awk 'NR==2 {print $5}' | sed 's/%//')

if [ "$disk_kullanim" -ge 90 ]; then
    renk=$KIRMIZI
elif [ "$disk_kullanim" -ge 70 ]; then
    renk=$SARI
else
    renk=$YESIL
fi

echo -e "Kök Dizin Doluluk: ${renk}%${disk_kullanim}${NORMAL}"

# --- 3. RAM Kullanımı ---
baslik_yaz "RAM KULLANIMI"
```

```
# free -m çıktısını parse ediyoruz
toplam_ram=$(free -m | awk '/Mem:/ {print $2}')
kullanilan_ram=$(free -m | awk '/Mem:/ {print $3}')
bos_ram=$(free -m | awk '/Mem:/ {print $4}')

echo "Toplam RAM    : ${toplam_ram} MB"
echo "Kullanılan    : ${kullanilan_ram} MB"
echo "Boş           : ${bos_ram} MB"

# --- 4. Sonuç ---
echo -e "\n${YESIL}Rapor başarıyla tamamlandı.${NORMAL}"
```

Nasıl Çalıştırılır? 1. Dosyayı kaydedin: `nano sistem_raporu.sh` 2. İzin verin: `chmod +x sistem_raporu.sh` 3. Çalıştırın: `./sistem_raporu.sh`

12.2. Proje 2: Otomatik Dosya/Dizin Yedekleme Scripti

Bu projede, belirli bir klasörü sıkıştırıp (`tar.gz`), ismine tarih ekleyerek yedekleyen ve işlemi loglayan bir otomasyon scripti yazacağız.

Amaç: * Argüman kontrolü yapmak (Hangi klasör yedeklenecek?). * `tar` komutu ile sıkıştırma. * Loglama fonksiyonu kullanmak. * Hata yönetimi (`trap`).

Script Kodu (`yedekle.sh`):

```
#!/bin/bash

# --- Ayarlar ---
YEDEK_DIZINI="/home/ender/yedekler"
LOG_DOSYASI="/var/log/yedekleme.log"
ZAMAN_DAMGASI=$(date "+%Y%m%d_%H%M%S")

# --- Log Fonksiyonu ---
logla() {
    local mesaj="$1"
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $mesaj" >> "$LOG_DOSYASI"
    echo "$mesaj" # Ekrana da basalım
}

# --- Hazırlık ---
# Yedeklerin konacağı klasör yoksa oluştur
mkdir -p "$YEDEK_DIZINI"

# Kullanıcı klasör adı girmemişse uyar
if [ -z "$1" ]; then
    echo "Kullanım: $0 <yedeklenecek_klasor_yolu>"
    exit 1
fi
```

```
HEDEF_KLASOR="$1"

# Hedef klasör gerçekten var mı?
if [ ! -d "$HEDEF_KLASOR" ]; then
    echo "Hata: '$HEDEF_KLASOR' bir klasör değil veya bulunamadı!"
    exit 1
fi

# Klasör adını temizleyelim (/home/user/data -> data)
klasor_adi=$(basename "$HEDEF_KLASOR")
ARSIV_ADI="${YEDEK_DIZINI}/${klasor_adi}_${ZAMAN_DAMGASI}.tar.gz"

# --- İşlem Başlıyor ---
logla "Yedekleme başladı: $HEDEF_KLASOR"

# tar komutu ile sıkıştır (-c: create, -z: gzip, -f: file)
# 2> /dev/null ile hata mesajlarını gizleyebiliriz ama loglamak daha iyidir.
if tar -czf "$ARSIV_ADI" "$HEDEF_KLASOR" 2>> "$LOG_DOSYASI"; then
    logla "Yedekleme BAŞARILI: $ARSIV_ADI"

    # Dosya boyutunu da yazalım
    boyut=$(du -h "$ARSIV_ADI" | awk '{print $1}')
    logla "Yedek Boyutu: $boyut"
else
    logla "[HATA] Yedekleme sırasında sorun oluştu!"
    exit 1
fi
```

Cron Entegrasyonu: Bu scripti her gece çalıştırmak için: `0 3 * * * /home/ender/scripts/yedekle.sh /var/www/html`

12.3. Proje 3: Log Dosyası Analiz Aracı

Bu projede, bir web sunucusunun (Apache/Nginx) erişim loglarını okuyup, en çok ziyaret edilen ip adreslerini ve hata kodlarını (404, 500) listeleyen bir analiz aracı yapacağız.

Amaç: * `awk`, `sort`, `uniq` komutlarını etkin kullanmak. * Pipe (`|`) zincirleme işlemlerini pekiştirmek.

Örnek Log Verisi (`access.log`):

```
192.168.1.10 - - [30/Dec/2025:10:00:01] "GET /index.html HTTP/1.1" 200 1024
192.168.1.15 - - [30/Dec/2025:10:05:01] "GET /admin.php HTTP/1.1" 404 512
192.168.1.10 - - [30/Dec/2025:10:10:01] "POST /login HTTP/1.1" 500 2048
```

Script Kodu (`log_analiz.sh`):

```
#!/bin/bash
```

```
LOG_DOSYASI=${1:-access.log}

if [ ! -f "$LOG_DOSYASI" ]; then
    echo "Hata: Log dosyası bulunamadı ($LOG_DOSYASI)"
    exit 1
fi

echo "=== LOG ANALİZ RAPORU: $LOG_DOSYASI ==="

# --- 1. En Çok İstek Yapan IP Adresleri (İlk 5) ---
echo -e "\n[TOP 5 IP Adresleri]"
# awk '{print $1}': 1. sütundaki IP'leri al
# sort: Sırala (uniq için gerekli)
# uniq -c: Tekrarları say (count)
# sort -nr: Sayıya göre ters (büyükten küçüğe) sırala
# head -n 5: İlk 5'i göster
awk '{print $1}' "$LOG_DOSYASI" | sort | uniq -c | sort -nr | head -n 5

# --- 2. HTTP Durum Kodları (200, 404 vs) ---
echo -e "\n[HTTP Durum Kodu Dağılımı]"
# Genellikle HTTP kodu 9. sütundadır
awk '{print $9}' "$LOG_DOSYASI" | sort | uniq -c | sort -nr

# --- 3. 404 Hataları (Sayfa Bulunamadı) ---
echo -e "\n[404 Hataları]"
# HTTP kodu 404 olan satırları bul ve say
hata_sayisi=$(grep " 404 " "$LOG_DOSYASI" | wc -l)
echo "Toplam 404 Hatası: $hata_sayisi"

# --- 4. Potansiyel Saldırganlar (Çok Fazla 404 Alanlar) ---
echo -e "\n[Şüpheli IP'ler (404 Alanlar)]"
grep " 404 " "$LOG_DOSYASI" | awk '{print $1}' | sort | uniq -c | sort -nr | head -3
```

12.4. Proje 4: Kullanıcı Yönetim Otomasyonu

Bu projede, `kullanicilar.txt` adında bir dosyadan "KullanıcıAdı:Departman" formatındaki verileri okuyup, bu kullanıcıları sisteme ekleyen, onlara rastgele bir parola oluşturan ve bunları bir rapora yazan script hazırlayacağız.

Amaç: * Dosya okuma döngüsü (`while read`). * String manipülasyonu (Bölme işlemleri). * Rastgele veri üretimi (`$RANDOM` veya `openssl`). * Global komutlar (`useradd`, `chpasswd`).

Girdi Dosyası (`kullanicilar.txt`):

```
ahmet:IT
ayse:IK
mehmet:Finans
```

Script Kodu (`kullanici_ekle.sh`):

```
#!/bin/bash

DOSYA="kullanicilar.txt"
CIKTI_DOSYASI="yeni_kullanicilar.csv"

# Root kontrolü (Kullanıcı eklemek için root olmak gerekir)
if [ "$EUID" -ne 0 ]; then
    echo "Lütfen bu scripti root yetkisiyle çalıştırın (sudo)."
    exit 1
fi

echo "Kullanıcı Adı,Parola,Departman" > "$CIKTI_DOSYASI"

while read -r satir; do
    # Satır boşsa atla
    [ -z "$satir" ] && continue

    # Veriyi ':' karakterine göre böl
    # IFS (Internal Field Separator) kullanarak değişkenlere atama
    IFS=':' read -r kullanici departman <<< "$satir"

    # Kullanıcı zaten var mı?
    if id "$kullanici" &>/dev/null; then
        echo "$kullanici zaten mevcut, atlanıyor."
        continue
    fi

    echo "Ekleniyor: $kullanici ($departman)"

    # 1. Kullanıcıyı oluştur (-m: home dizini oluştur, -s: shell belirle)
    useradd -m -s /bin/bash "$kullanici"

    # 2. Rastgele parola oluştur (openssl ile)
    parola=$(openssl rand -base64 12)

    # 3. Parolayı ayarla
    echo "$kullanici:$parola" | chpasswd

    # 4. Bilgileri CSV dosyasına kaydet
    echo "$kullanici,$parola,$departman" >> "$CIKTI_DOSYASI"

done < "$DOSYA"

echo -e "\nİşlem tamamlandı! Şifreler '$CIKTI_DOSYASI' dosyasında."
```

Güvenlik Uyarısı: Scriptin ürettiği `yeni_kullanicilar.csv` dosyasında parolalar açık metin olarak yazar. İşiniz bitince bu dosyayı güvenli bir şekilde silmeyi veya şifrelemeyi unutmayın!

Appendix A: Ek A: Terminal Kısayolları

Bash terminalini bir usta gibi kullanmak için farenizi bırakın ve bu kısayolları ezberleyin. Hızınızı 10 kat artıracaktır.

A.1. İmleç Hareketi

- **CTRL** + **A** : Satır başını gider.
- **CTRL** + **E** : Satır sonuna gider.
- **ALT** + **B** : Bir kelime geri gider (Back).
- **ALT** + **F** : Bir kelime ileri gider (Forward).
- **CTRL** + **XX** : İmleç ile satır başı arasında gidip gelir.

A.2. Düzenleme ve Silme

- **CTRL** + **U** : İmleçten satır başına kadar her şeyi siler (Hatalı yazdığınızda hayat kurtarır).
- **CTRL** + **K** : İmleçten satır sonuna kadar her şeyi siler.
- **CTRL** + **W** : İmleçten geriye doğru bir kelime siler.
- **CTRL** + **L** : Ekranı temizler (`clear` komutuyla aynıdır).

A.3. Geçmiş ve Arama

- **CTRL** + **R** : Geçmişte komut arar (Reverse Search). Yazmaya başladığınızda eşleşen en son komutu bulur. Tekrar basarsanız daha eskiye gider.
- **Up / Down Okları**: Geçmişteki komutları sırayla getirir.
- **!!** : Son çalıştırılan komutu tekrar eder (Örn: `sudo !!`).

A.4. Süreç Kontrolü

- **CTRL** + **C** : Çalışan komutu iptal eder/öldürür (SIGINT).
- **CTRL** + **Z** : Çalışan komutu durdurur ve arka plana atar (SIGSTOP). (`fg` ile geri alabilirsiniz).
- **CTRL** + **D** : Terminalden çıkış yapar (`exit` komutuyla aynıdır).

Appendix B: Ek B: Genişletilmiş Linux Komutları Referansı

Bash script yazarken veya sunucu yönetirken elinizin altında bulunması gereken **temel ve ileri seviye** komutların kapsamlı listesi.

B.1. 1. Dosya ve Dizin Yönetimi

Dosya sisteminde gezinmek ve düzenleme yapmak için gerekenler.

- `ls -lah` : Tüm dosyları (gizliler dahil), boyutları ve izinleriyle listeler.
- `cd -` : Bir önceki bulunduğunuz klasöre geri döner (Geri tuşu gibi).
- `mkdir -p a/b/c` : İç içe klasör ağacı oluşturur.
- `cp -r kaynak hedef` : Klasörleri içeriğiyle birlikte kopyalar.
- `mv dosya yeni_ad` : Dosya taşır veya yeniden adlandırır.
- `rm -rf klasor` : Klasörü ve içindekileri **sorgusuz sualsiz** siler.
- `ln -s dosya link` : Dosyaya sembolik link (kısayol) oluşturur.
- `file dosya` : Dosyanın türünü (resim, metin, script vb.) söyler.

B.2. 2. Dosya Arama ve Bulma

- `find / -name "log*"` : Tüm sistemde ismi log ile başlayan dosyaları arar.
- `find . -type f -size +100M` : 100MB'dan büyük dosyaları bulur.
- `grep -r "hata" .` : Bulunduğunuz klasördeki tüm dosyaların İÇİNDE "hata" kelimesini arar.
- `locate dosya` : Dosyayı veritabanından ışık hızında bulur (önce `updatedb` yapmak gerekir).
- `which komut` : Bir komutun hangi klasörde çalıştığını gösterir (Örn: `which python`).

B.3. 3. Arşivleme ve Sıkıştırma

- `tar -czf arsiv.tar.gz klasor` : Klasörü `.tar.gz` olarak sıkıştırır.
- `tar -xzf arsiv.tar.gz` : `.tar.gz` dosyasını olduğu yere çıkarır.
- `zip -r dosya.zip klasor` : Klasörü zip formatında sıkıştırır.
- `unzip dosya.zip` : Zip dosyasını açar.

B.4. 4. Sistem İzleme ve Performans

- `top` / `htop` : İşlemci ve RAM kullanımını canlı gösterir. (`htop` daha renklidir).
- `df -h` : Disk doluluk oranlarını (GB/TB cinsinden) gösterir.
- `du -sh *` : Bulunduğunuz klasördeki her bir dosyanın/klasörün boyutunu gösterir.

- `free -m` : RAM durumunu MB cinsinden gösterir.
- `uptime` : Sistemin ne kadar süredir açık olduğunu gösterir.
- `dmesg` : Çekirdek (kernel) ve donanım mesajlarını görüntüler.

B.5. 5. Ağ ve Bağlantı (Networking)

- `ip addr` : IP adresini gösterir.
- `ping -c 4 google.com` : Bağlantıyı test eder (4 paket atıp durur).
- `curl -I https://site.com` : Sitenin HTTP başlık bilgilerini çeker.
- `wget <url>` : Dosya indirir.
- `netstat -tulpn` : Hangi portun hangi program tarafından dinlendiğini gösterir.
- `ss -tuln` : `netstat'ın modern halidir, açık portları gösterir.
- `dig site.com` : DNS sorgusu yapar.

B.6. 6. Uzak Bağlantı ve Transfer

- `ssh user@ip` : Uzak sunucuya bağlanır.
- `scp dosya user@ip:/hedef` : Dosyayı güvenli şekilde karşı sunucuya kopyalar.
- `rsync -avz kaynak hedef` : Dosyaları akıllıca senkronize eder (Sadece değişenleri atar).

B.7. 7. Kullanıcı ve İzinler

- `sudo command` : Komutu yönetici (root) yetkisiyle çalıştırır.
- `chmod 755 dosya` : Çalıştırma izni verir.
- `chown user:group dosya` : Dosyanın sahibini değiştirir.
- `useradd -m yeniuser` : Yeni kullanıcı oluşturur.
- `passwd user` : Kullanıcının şifresini değiştirir.
- `whoami` : Hangi kullanıcı ile bağlı olduğunuzu söyler.
- `last` : Sisteme en son kimlerin girdiğini listeler.

B.8. 8. Süreç (Process) Yönetimi

- `ps aux | grep java` : Çalışan Java uygulamalarını bulur.
- `kill <PID>` : PID numarası verilen uygulamayı nazikçe kapatır.
- `kill -9 <PID>` : Uygulamayı zorla öldürür (Force Kill).
- `killall firefox` : İsminden tüm Firefox süreçlerini kapatır.
- `bg` / `fg` : Arka plandaki işi öne alır veya tam tersi.