Linux Shell Script Nedir?

Linux Shell Script, Linux veya Unix tabanlı işletim sistemlerinde, kabuk (shell) adı verilen bir arayüz üzerinde komutların sıralı bir şekilde çalıştırılmasını sağlayan betik dosyalarıdır. Bir shell script, genellikle aşağıdaki işlevleri yerine getirir:

→ Komutların otomasyonu: Kullanıcı tarafından verilen bir dizi komutu toplu şekilde çalıştırır.

→ İş süreçlerini yönetme: Dosya işlemleri, sistem kontrolü, kullanıcı girişi ve çıkışı, veri işleme gibi işlemleri gerçekleştirme.

→ Programlama mantığı içerme: Döngüler, koşullu ifadeler, değişkenler, fonksiyonlar gibi programlama yapıları içerir.

Shell scriptleri, .sh uzantısıyla kaydedilebilir ve belirli izinler verilerek çalıştırılabilir hale getirilebilir.

Linux Shell'in Tarihi

Unix ve Başlangıçlar (1960'lar sonu - 1970'ler)

- Linux Shell'in tarihi, Unix işletim sistemine dayanır. 1969 yılında Bell Labs'de Dennis Ritchie ve Ken Thompson, Unix'i geliştirirken komut satırında kullanıcılarla etkileşim kurmak için **Thompson Shell (sh)** adı verilen basit bir kabuk yazdılar.

Bourne Shell (1979)

- Unix sistemlerinin popülerleşmesiyle birlikte, daha gelişmiş bir kabuk ihtiyacı doğdu. **Stephen Bourne**, 1979'da AT&T Bell Labs'de **Bourne Shell (sh)**'i geliştirdi. Bu kabuk, bugünkü birçok shell script'in temelini oluşturur. Bourne Shell, programlama yapıları (döngüler, koşullar, fonksiyonlar) sunarak script yazımını kolaylaştırdı.

C Shell (csh) ve Korn Shell (ksh) (1980’ler)

- 1980’lerde, Unix üzerinde farklı kabuklar ortaya çıktı. **C Shell (csh)**, C programlama diline benzeyen bir sözdizimi sundu ve özellikle programcılar arasında popüler oldu.
- **Korn Shell (ksh)**, **David Korn** tarafından geliştirildi ve Bourne Shell'in yeteneklerini geliştirerek performans ve programlama özellikleri açısından daha zengin bir yapı sağladı.

Bash (1989)

- **Bash (Bourne Again Shell)**, 1989 yılında **Brian Fox** tarafından GNU projesi kapsamında geliştirildi. Bash, Bourne Shell’in yerine geçmek üzere tasarlandı ve hem Bourne Shell'in hem de C Shell'in özelliklerini birleştirerek çok güçlü bir kabuk haline geldi.
- Bugün Linux dağıtımlarında varsayılan shell genellikle Bash’tir.

Shell Script'te Değişkenler (Variables)

Değişkenler, shell script'lerde değerlerin saklanmasını ve kullanılmasını sağlayan yapılar olup, herhangi bir değeri (sayı, metin, komut çıktısı vb.) tutabilirler. Shell script'te değişkenler, komut satırında çalıştırılan komutlar ile etkileşime geçmek, kullanıcı girdisi almak veya dinamik işlemler yapmak için yaygın olarak kullanılır.

1. Değişken Tanımlama

Shell script'te değişkenler, sadece bir isim ve değer ataması ile tanımlanır. Eşittir (=) operatörü kullanılarak değişkene bir değer atanır. Değişken adının önüne $ işareti eklenerek değeri çağrılır. Bash ve diğer shell script'lerde değişken isimlerinde boşluk kullanılmaz.

```
VARIABLE_NAME=value
```

```
#!/bin/bash
NAME="Ahmet"
echo "Merhaba, $NAME"
```

Bu örnekte, **NAME** değişkenine **"Ahmet"** değeri atanmış ve echo komutu ile ekrana yazdırılmıştır. Çıktı şu olacaktır:

```
Merhaba, Ahmet
```

2.Değişkenlerde İki Tip Kapsam Vardır

→ Yerel (Local) Değişkenler: Script içerisinde geçici olan, sadece o script içerisinde kullanılabilen değişkenlerdir. Shell kapandığında ya da script bittiğinde bu değişkenler kaybolur.

→ Ortam (Environment) Değişkenleri: Shell'in başlarken tanımladığı ve sistem boyunca kullanılabilen değişkenlerdir. Ortam değişkenleri **export** komutu ile tanımlanabilir.

```
#!/bin/bash
MYVAR="Yerel değişken"
export MYENVVAR="Ortam değişkeni"
echo "Yerel değişken: $MYVAR"
echo "Ortam değişkeni: $MYENVVAR"
```

3.Değişkenlerde Sabit Değer (Constant) Tanımlama

Shell script'te sabit değerler (const) için belirli bir yapı olmasa da, değişkenin başına readonly ekleyerek değişkenin değeri değiştirilemez hale getirilebilir.

```
readonly PI=3.14
```

Bu değişkenin değerini değiştirme girişimi hata verecektir.

4.Matematiksel İşlemler

Shell script'te değişkenler arasında matematiksel işlemler yapmak için genellikle **expr** veya **$(( ))** kullanılır. expr komutu, matematiksel ifadeleri işlemek için kullanılır.

```
#!/bin/bash
a=5
b=3
toplam=$((a + b))
echo "Toplam: $toplam"
```

5.Komut Çıktısını Değişkene Atama

Bir komutun çıktısını değişkene atamak için ters tırnak (`) ya da **$(komut)** yapısı kullanılır.

```
#!/bin/bash
TARIH=$(date)
echo "Bugünün tarihi: $TARIH"
```

Burada, **date** komutunun çıktısı **TARIH** değişkenine atanır.

6.Diziler

Bash script'te diziler birden fazla değeri tutabilir. Dizilerde indeksler 0'dan başlar.

```
#!/bin/bash
MYARRAY=("elma" "armut" "portakal")
echo "İlk eleman: ${MYARRAY[0]}"
```

Ön Tanımlı Değişkenler

Shell script'lerde kullanılan ön tanımlı değişkenler, sistem tarafından otomatik olarak atanır ve belirli bir işlevi yerine getirmek için kullanılır. Bu değişkenler, genellikle script'in çalışma durumu, argümanlar ve süreç yönetimi hakkında bilgi sağlar.

1. $0: Script'in Adı

$0 değişkeni, çalıştırılan script'in adını döndürür.

```
#!/bin/bash
echo "Çalışan script'in adı: $0"
```

Çıktı:

```
Çalışan script'in adı: ./script_adi.sh
```

Bu örnekte, **$0**, script'in kendisinin adını temsil eder. Script, nerede ve nasıl çalıştırılırsa çalıştırılsın, adı **$0** ile görüntülenebilir.

2. $1, $2, ..., $N: Script'e Verilen Argümanlar

$1 ilk argümanı, **$2** ikinci argümanı ve bu şekilde **$N** argümanlarını temsil eder.

```
#!/bin/bash
echo "Birinci argüman: $1"
echo "İkinci argüman: $2"
```

Çalıştırma:

```
./script.sh Ahmet Merhaba
```

Çıktı:

```
Birinci argüman: Ahmet
İkinci argüman: Merhaba
```

Bu şekilde, komut satırında verilen argümanlar sırasıyla $1, $2 gibi değişkenler aracılığıyla kullanılır.

3. $#: Argüman Sayısı

$# değişkeni, script'e kaç tane argüman verildiğini gösterir.

```
#!/bin/bash
echo "Toplam argüman sayısı: $#"
```

Çalıştırma:

```
./script.sh Ahmet Merhaba
```

Çıktı:

```
Toplam argüman sayısı: 2
```

4. $@: Tüm Argümanlar

$@, tüm argümanları bir dizi gibi listeler ve argümanları ayrı ayrı işler. Döngü veya işlem yaparken argümanları bu değişkenle işleyebilirsiniz.

```
#!/bin/bash
for arg in "$@"
do
    echo "Argüman: $arg"
done
```

Çalıştırma:

```
./script.sh Ahmet Merhaba Dünya
```

Çıktı:

```
Argüman: Ahmet
Argüman: Merhaba
Argüman: Dünya
```

5. $*: Tüm Argümanlar (Tek String Olarak)

$*, tüm argümanları tek bir string olarak döndürür. Argümanlar bir bütün olarak değerlendirilir, boşluklarla ayrılır.

```
#!/bin/bash
echo "Tüm argümanlar: $*"
```

Çalıştırma:

```
./script.sh Ahmet Merhaba Dünya
```

Çıktı:

```
Tüm argümanlar: Ahmet Merhaba Dünya
```

$@** ile **$* arasındaki fark şudur: **$@** argümanları ayrı ayrı işlerken, **$*** hepsini tek bir string olarak işler.

6. $?: Son Komutun Çıkış Durumu

$? değişkeni, son çalıştırılan komutun çıkış durumunu döndürür. Komut başarılı bir şekilde çalıştıysa **0**, bir hata varsa hata kodu döner.

```
#!/bin/bash
ls /var/log
echo "Çıkış durumu: $?"
```

Başarılı Çalıştırma:

```
Çıkış durumu: 0
```

Hatalı Çalıştırma(Geçersiz Dizin,):

```
ls: cannot access '/var/invalid': No such file or directory
Çıkış durumu: 2
```

7. $$: Script'in PID'si (İşlem Kimliği)

$$, çalışmakta olan script'in **Process ID (PID)** numarasını döndürür. Bu, script'in işlem kimliğini elde etmek için kullanılır.

```
#!/bin/bash
echo "Script'in PID'si: $$"
```

Çıktı:

```
Script'in PID'si: 12345
```

PID, bir işlemi tanımlayan benzersiz numaradır. Shell script'lerde bu PID numarası, arka planda çalışan iş süreçlerini takip etmek için faydalıdır.

Shell Script'te Koşullu İfadeler (Conditional statements (if/else))

Shell script'lerde **if** yapısı, koşullu ifadeleri kontrol etmek ve belirli koşullar sağlandığında farklı işlemler yapabilmek için kullanılır. **if** yapısı, programlama dillerindeki diğer **if** yapılarıyla oldukça benzerdir. Koşullar sağlandığında (**true** olduğunda) belirli bir kod bloğu çalıştırılır; koşullar sağlanmadığında (**false** olduğunda) başka bir kod bloğu çalıştırılabilir.

1. Temel if Yapısı

Shell script'te if yapısının temel biçimi şu şekildedir:

```
if [ koşul ]
then
    # Koşul doğruysa (true) burası çalışır
    komutlar
fi
```

if [koşul] → Koşulu kontrol eder. Eğer koşul doğruysa (true), then kısmındaki komutlar çalışır.

then → Koşul doğru olduğunda çalıştırılacak komutların başladığı yerdir.

fi → if yapısını sonlandıran anahtar kelimedir (tersi if gibi düşünülebilir).

2. if-else Yapısı

Eğer koşul sağlanmazsa (false dönerse), alternatif bir yol olan else kısmı devreye girebilir. Bu, programın alternatif bir komut bloğunu çalıştırmasına olanak tanır.

```
if [ koşul ]
then
    # Koşul doğruysa (true) burası çalışır
    komutlar
else
    # Koşul yanlışsa (false) burası çalışır
    komutlar
fi
```

3. if-elif-else Yapısı

Birden fazla koşulu kontrol etmek için elif (else if) yapısı kullanılır. Bu, birden fazla koşulu sırayla kontrol etmeyi sağlar. İlk doğru (true) koşul bulunduğunda o bloğa girilir ve diğer koşullar kontrol edilmez.

```
if [ koşul1 ]
then
    # Koşul1 doğruysa burası çalışır
    komutlar
elif [ koşul2 ]
then
    # Koşul2 doğruysa burası çalışır
    komutlar
else
    # Hiçbir koşul doğru değilse burası çalışır
    komutlar
fi
```

4. Koşul Kontrolleri

Karşılaştırma Operatörleri:

-eq → İki sayının eşit olup olmadığını kontrol eder.
-ne → İki sayının eşit olmadığını kontrol eder.
-gt → İlk sayı ikinci sayıdan büyükse.
-lt → İlk sayı ikinci sayıdan küçükse.
-ge → İlk sayı ikinci sayıdan büyük veya eşitse.
-le → İlk sayı ikinci sayıdan küçük veya eşitse.

Dosya Kontrolleri:

-e → Dosya veya dizin var mı?
-f → Dosya var mı ve dosya mı?
-d → Dizin var mı?
-r → Dosya okuma iznine sahip mi?
-w → Dosya yazma iznine sahip mi?
-x → Dosya çalıştırma iznine sahip mi?

Metin Karşılaştırmaları:

= → İki string eşit mi?
!= → İki string farklı mı?
-z → String'in uzunluğu sıfır mı (boş mu)?
-n → String'in uzunluğu sıfır değil mi (dolu mu)?

```
#!/bin/bash

echo "Bir sayı girin:"
read sayi

if [ "$sayi" -gt 10 ]
then
    echo "Sayı 10'dan büyük."
else
    echo "Sayı 10'dan küçük veya eşit."
fi
```

Bu örnekte, kullanıcının girdiği sayı 10'dan büyükse bir mesaj yazdırılır, değilse başka bir mesaj yazdırılır.

NOT : **Read** komutunu kullanıcıdan bir değer alabilmek için kullanırız. Yukarıdaki örnekte kullanıcı klavyeden bir sayı girer ve girdiği bu değer “sayi” değişkenine atanır.

```
#!/bin/bash

echo "Kontrol etmek istediğiniz dosya yolunu girin:"
read dosya

if [ -e "$dosya" ]
then
    echo "Dosya var."
else
    echo "Dosya yok."
fi
```

Bu örnekte, kullanıcıdan alınan dosya yolunun var olup olmadığı kontrol edilir. Eğer dosya mevcutsa "Dosya var" mesajı yazdırılır, değilse "Dosya yok" mesajı yazdırılır.

```
#!/bin/bash

echo "Bir kelime girin:"
read kelime

if [ "$kelime" = "merhaba" ]
then
    echo "Merhaba dünya!"
else
    echo "Başka bir şey girdiniz."
fi
```

Kullanıcıdan alınan kelimenin "merhaba" olup olmadığı kontrol edilir. Eğer doğruysa belirli bir mesaj yazdırılır.

```bash
#!/bin/bash

echo "Bir sayı girin:"
read sayi

if [ "$sayi" -lt 0 ]
then
    echo "Negatif bir sayı girdiniz."
elif [ "$sayi" -eq 0 ]
then
    echo "Sıfır girdiniz."
else
    echo "Pozitif bir sayı girdiniz."
fi
```

Bu örnekte, girilen sayının negatif, sıfır ya da pozitif olup olmadığı kontrol edilerek farklı mesajlar yazdırılır.

5. Mantıksal Operatörler

Shell script'te birden fazla koşulu aynı anda kontrol etmek için mantıksal operatörler kullanılır.

&& (ve) → Her iki koşul doğruysa çalışır.
|| (veya) → Koşullardan biri doğruysa çalışır.

```bash
#!/bin/bash

echo "Bir sayı girin:"
read sayi

if [ "$sayi" -gt 0 ] && [ "$sayi" -lt 100 ]
then
    echo "Sayı 0 ile 100 arasında."
else
    echo "Sayı 0 ile 100 arasında değil."
fi
```

Bu örnekte, kullanıcının girdiği sayının 0 ile 100 arasında olup olmadığı kontrol edilir.

6. Çift Köşeli Parantez [[ ]] Kullanımı

Çift köşeli parantezler **[[ ]]**, tek köşeli parantezlere göre daha fazla özellik sağlar. Örneğin, **==** operatörünü kullanarak string karşılaştırması yapmak, regex desteği sunmak ve daha gelişmiş kontroller yapmak mümkündür.

```bash
#!/bin/bash

echo "Bir kelime girin:"
read kelime

if [[ "$kelime" == "merhaba" ]]
then
    echo "Kelime merhaba!"
else
    echo "Kelime farklı!"
fi
```

Shell Script'te read Komutu

Shell script'te **read** komutu, kullanıcıdan veri almak için kullanılır. **read** komutu, komut satırında kullanıcıdan girdi isteyip, bu girdiği bir değişkene atayarak shell script'in daha dinamik olmasını sağlar.

1. Temel read Kullanımı

read komutu ile kullanıcıdan veri almak oldukça basittir. **read** komutunun hemen ardından bir değişken adı yazılır ve kullanıcı bir şeyler girdiğinde bu girdi değişkene atanır.

```
#!/bin/bash
echo "Adınızı girin:"
read isim
echo "Merhaba, $isim!"
```

read isim → Kullanıcıdan adını girmesi istenir ve girdiği ad isim değişkenine atanır.

echo "Merhaba, $isim!" → Girilen ad ekrana yazdırılır.

2. Birden Fazla Değişkene Veri Atama

read komutuyla aynı anda birden fazla değişkene veri atayabilirsiniz. Kullanıcı, girdiğinde her boşluk bir sonraki değişkene atanır.

```
#!/bin/bash
echo "Adınızı ve soyadınızı girin:"
read isim soyisim
echo "Adınız: $isim"
echo "Soyadınız: $soyisim"
```

İlk girilen değer **isim** değişkenine, ikinci değer **soyisim** değişkenine atanır.

3. Kullanıcıdan Girdi Alırken İsteği Bir Mesajla Gösterme

read komutu ile kullanıcıdan veri alırken, prompt (istem) mesajı kullanarak ne tür bir bilgi istediğinizi belirtebilirsiniz. Bu, **-p** parametresi ile yapılır.

```
#!/bin/bash
read -p "Adınızı girin: " isim
echo "Merhaba, $isim!"
```

Bu örnekte **-p** parametresi sayesinde kullanıcıya bir istem mesajı gösterilir ve **read** komutu ile girdiyi alır.

4. Veriyi Gizlemek (Şifre Girişi)

Bazı durumlarda, kullanıcıdan gizli bir bilgi (örneğin, şifre) almak isteyebilirsiniz. **-s** parametresi ile kullanıcının girdiği metni gizleyebilirsiniz.

```
#!/bin/bash
read -sp "Şifrenizi girin: " sifre
echo
echo "Şifre alındı!"
```

→ **-s** parametresi ile kullanıcı girdisi ekranda görünmez.

→ **echo** komutu ile boş bir satır yazdırılır (şifre girişi sırasında girdiği yazı gözükmediği için bir boşluk bırakılır).

5. Varsayılan Girdi Kullanma

Eğer kullanıcı hiçbir girdi yapmazsa, **read** komutu varsayılan bir değeri kabul edebilir. Bu işlem, değişkeni önceden tanımlayarak yapılır.

```
#!/bin/bash
isim="Anonim"
read -p "Adınızı girin: " kullanici_girdisi
isim=${kullanici_girdisi:-$isim}
echo "Merhaba, $isim!"
```

→ Kullanıcı hiçbir şey girmezse, **isim** değişkenine **"Anonim"** değeri atanır.

→ Eğer kullanıcı bir giriş yaparsa, girilen değer **isim** değişkenine atanır.

6. Bir Dosyadan Veri Okuma (Readline Modu)

read komutu bir dosyadan satır satır veri okuyabilir. Bu özellikle dosyalardaki bilgileri işlemek için kullanılır.

```
#!/bin/bash
while read satir
do
    echo "Okunan satır: $satir"
done < dosya.txt
```

→ **while read** döngüsü, **dosya.txt** içindeki her bir satırı sırayla okur ve **satir** değişkenine atar.

→ Her satır **echo** komutu ile ekrana yazdırılır.

NOT: While döngüsünden bahsedilecek.

7. İnput'ları Bir Diziye Atama

read komutu, tüm girdileri bir diziye atayarak da kullanılabilir.

```
#!/bin/bash
echo "İsimleri girin (boşluklarla ayırın):"
read -a isimler
echo "Girdiğiniz isimler:"
for isim in "${isimler[@]}"
do
    echo $isim
done
```

→ **-a** parametresi, girdiyi bir dizi olarak alır.

→ Kullanıcının girdiği her bir isim diziye atanır ve sonrasında döngü ile her biri ekrana yazdırılır.

NOT: For döngüsünden bahsedilecek.

Shell Script'te while Döngüsü

while döngüsü, belirli bir koşul doğru olduğu sürece (koşul **true** olduğu sürece) bir dizi komutun tekrar tekrar çalıştırılmasını sağlar. Koşul **false** olduğunda döngü sonlanır. Bu döngü, genellikle tekrar eden görevleri yerine getirmek veya belirli bir duruma kadar beklemek için kullanılır.

1. Temel while Döngüsü Yapısı

while döngüsünün temel yapısı şu şekildedir:

```
while [ koşul ]
do
    # Koşul doğru olduğu sürece bu komutlar çalışır
    komutlar
done
```

while [koşul] → Koşul doğru olduğu sürece do bloğu içindeki komutlar çalıştırılır.

do → Koşul doğruysa çalıştırılacak komutları belirtir.

done → Döngünün sonunu belirtir.

Örnek:

```
#!/bin/bash

sayi=1

while [ $sayi -le 5 ]
do
    echo "Sayı: $sayi"
    sayi=$((sayi + 1))
done
```

Çıktı:

```
Sayı: 1
Sayı: 2
Sayı: 3
Sayı: 4
Sayı: 5
```

→ **Döngü**, sayi değişkeni 5'ten küçük veya eşit olduğu sürece (-le 5) devam eder.

→ Her döngüde **echo** komutu ile sayı yazdırılır ve ardından **sayi** değişkeni 1 artırılır.

2.Kullanıcıdan Veri Alarak Döngü

Bir **while** döngüsü, kullanıcıdan veri alarak çalışabilir ve koşul kullanıcı girişine göre belirlenebilir. Örneğin, kullanıcıdan doğru bir değer girene kadar döngü devam edebilir.

```
#!/bin/bash

sifre="1234"
giris=""

while [ "$giris" != "$sifre" ]
do
    read -p "Şifreyi girin: " giris
    if [ "$giris" != "$sifre" ]; then
        echo "Hatalı şifre!"
    fi
done

echo "Doğru şifre girildi."
```

→ Kullanıcı doğru şifreyi girene kadar (**giris != sifre** olduğu sürece) döngü devam eder.

→ Her hatalı girişte "Hatalı şifre!" mesajı görüntülenir.

→ Doğru şifre girildiğinde döngü sona erer ve "Doğru şifre girildi." mesajı yazdırılır.

3. Sonsuz Döngü

while döngüsü, eğer koşul her zaman doğru olacak şekilde tanımlanırsa, sonsuz bir döngüye dönüşebilir. Sonsuz döngü, belirli bir koşul sağlanmadığı sürece durmadan tekrar eder. Bu tür döngüler genellikle belirli bir işlem durdurulana kadar çalışmaya devam etmesi gereken görevler için kullanılır.

```
#!/bin/bash

while true
do
    echo "Bu bir sonsuz döngüdür. Çıkmak için Ctrl+C yapın."
    sleep 1
done
```

while true → Koşul her zaman doğru olduğundan döngü sonsuz kez tekrar eder.

sleep 1 → Her bir döngü arasında 1 saniye bekleme eklenmiştir. Bu, döngünün kontrolsüz şekilde çok hızlı çalışmasını önler.

→ Döngüden çıkmak için terminalde **Ctrl+C** tuşlarına basarak süreci durdurabilirsiniz.

4. break ve continue Komutları

Döngüler içinde **break** ve **continue** komutları kullanılarak döngünün akışı kontrol edilebilir.

break: Döngüyü tamamen sonlandırır.

continue: Döngünün o anki adımını atlar ve bir sonraki adıma geçer.

```
#!/bin/bash

sayi=1

while [ $sayi -le 10 ]
do
    if [ $sayi -eq 5 ]; then
        break
    fi
    echo "Sayı: $sayi"
    sayi=$((sayi + 1))
done

echo "Döngü 5'e ulaştı ve durduruldu."
```

→ **sayi** değişkeni 5'e ulaştığında **break** komutu çalışır ve döngü tamamen sonlanır.

Çıktı:

```
Sayı: 1
Sayı: 2
Sayı: 3
Sayı: 4
Döngü 5'e ulaştı ve durduruldu.
```

```bash
#!/bin/bash

sayi=0

while [ $sayi -le 5 ]
do
    sayi=$((sayi + 1))
    if [ $sayi -eq 3 ]; then
        continue
    fi
    echo "Sayı: $sayi"
done
```

sayi 3'e ulaştığında **continue** komutu çalışır ve o adım atlanır, sonraki sayıya geçilir.

Çıktı:

```
Sayı: 1
Sayı: 2
Sayı: 4
Sayı: 5
Sayı: 6
```

Shell Script'te for Döngüsü

Shell script'te **for** döngüsü, bir dizi üzerinde sırasıyla gezinmek ve her eleman için belirli komutları çalıştırmak için kullanılır. **for** döngüsü, liste ya da aralıktaki değerler üzerinde tekrarlı işlem yapmak için kullanılır.

1. Temel for Döngüsü Yapısı

Shell script'teki temel **for** döngüsü şu şekilde yapılandırılır:

```
for değişken in liste
do
    # Her bir eleman için çalışacak komutlar
    komutlar
done
```

değişken → Döngü her döndüğünde listenin bir sonraki elemanını tutar.

liste → Üzerinde döngü yapılacak öğelerin listesi (diziler, aralıklar, komut çıktıları vb.).

do → Döngü bloğunun başladığını belirtir.

done → Döngünün bittiğini belirtir.

2. Temel Örnek: Liste Üzerinde Dolaşma

Aşağıda bir kelime listesi üzerinde döngü işlemi gerçekleştiren basit bir **for** döngüsü örneği bulunmaktadır:

```
#!/bin/bash

for isim in Ahmet Ayşe Mehmet
do
    echo "İsim: $isim"
done
```

Çıktı:

```
İsim: Ahmet
İsim: Ayşe
İsim: Mehmet
```

→ **for** döngüsü, "Ahmet", "Ayşe" ve "Mehmet" değerlerini sırayla **isim** değişkenine atar ve her seferinde bu değeri **echo** komutu ile ekrana yazdırır.

3. Aralıklar Üzerinde Döngü

for döngüsü, belirli bir aralıktaki sayılar üzerinde de dönebilir. Bu, genellikle **seq** komutuyla kullanılır.

```
#!/bin/bash

for sayi in $(seq 1 5)
do
    echo "Sayı: $sayi"
done
```

Çıktı:

```
Sayı: 1
Sayı: 2
Sayı: 3
Sayı: 4
Sayı: 5
```

→ **seq 1 5** komutu, 1'den 5'e kadar olan sayıları üretir. Döngü her sayıyı sırayla **sayi** değişkenine atar ve yazdırır.

4. Diziler Üzerinde Döngü

Shell script'te diziler tanımlanıp, bu diziler üzerinde **for** döngüsü ile işlem yapılabilir.

```
#!/bin/bash

isimler=("Ali" "Veli" "Ayşe")

for isim in "${isimler[@]}"
do
    echo "İsim: $isim"
done
```

Çıktı:

```
İsim: Ali
İsim: Veli
İsim: Ayşe
```

→ **isimler** dizisi tanımlanmıştır. **for** döngüsü, dizi elemanları üzerinde sırayla döner ve her birini **isim** değişkenine atar.

5. Dosya veya Klasörler Üzerinde Döngü

for döngüsü ile bir dizindeki dosyalar veya klasörler üzerinde işlem yapmak mümkündür. Örneğin, belirli bir dizindeki tüm dosyaları listeleyebilirsiniz.

```
#!/bin/bash

for dosya in /home/kullanici/*
do
    echo "Dosya: $dosya"
done
```

```
Dosya: /home/kullanici/dosya1
Dosya: /home/kullanici/dosya2
...
```

/home/kullanici/*, bu dizindeki tüm dosyaları temsil eder. Döngü, dizindeki her dosyayı sırasıyla **dosya** değişkenine atar ve ekrana yazdırır.

6. Komut Çıktıları Üzerinde Döngü

for döngüsü, bir komutun çıktısını işlemek için de kullanılabilir. Örneğin, bir dizindeki dosyaları listeleyip bunlar üzerinde işlem yapabilirsiniz.

```
#!/bin/bash

for dosya in $(ls /home/kullanici)
do
    echo "Dosya: $dosya"
done
```

→ **ls /home/kullanici** komutu çalıştırılır ve çıktısı **for** döngüsünde kullanılır. Döngü her dosya veya klasör adını sırayla işler.

7.break ve continue Komutları ile Döngü Kontrolü

for döngüsü içinde **break** ve **continue** komutları kullanarak döngü akışını kontrol edebilirsiniz.

break: Döngüyü tamamen sonlandırır.

continue: Döngünün o anki adımını atlayarak bir sonraki adıma geçer.

```
#!/bin/bash

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

Çıktı:

```
Sayı: 1
Sayı: 2
Sayı: 3
Sayı: 4
```

→ Döngü 5'e ulaştığında **break** komutu çalışır ve döngü sonlanır.

```
#!/bin/bash

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

Çıktı:

```
Sayı: 1
Sayı: 2
Sayı: 4
Sayı: 5
```

→ 3 sayısına ulaşıldığında **continue** komutu çalışır ve o adım atlanır. Diğer sayılar yazdırılır.

Shell Script'te Fonksiyonlar

Shell script'te fonksiyonlar, belirli bir işlevi gerçekleştiren ve birden fazla kez çağrılabilen kod bloklarıdır. Fonksiyonlar, kod tekrarını azaltarak script'lerin daha modüler ve okunabilir olmasını sağlar. Bir fonksiyon tanımlandıktan sonra, script'in herhangi bir yerinde çağrılarak çalıştırılabilir.

1. Fonksiyonların Tanımlanması ve Kullanımı

Shell script'te bir fonksiyonun temel tanımı şu şekildedir:

```
function fonksiyon_adi {
    # Fonksiyonun içinde çalışacak komutlar
    komutlar
}
```

Veya fonksiyon şu şekilde de tanımlanabilir:

```
fonksiyon_adi() {
    # Fonksiyonun içinde çalışacak komutlar
    komutlar
}
```

Fonksiyon çağrıldığında içindeki komutlar çalıştırılır. Fonksiyonun içindeki kodlar, sadece fonksiyon çağrıldığında çalışır.

2. Basit Bir Fonksiyon Örneği

Aşağıdaki örnekte basit bir fonksiyon tanımlanmış ve ardından bu fonksiyon script içinde çağrılmıştır.

```
#!/bin/bash

# Fonksiyon tanımı
fonksiyon_adi() {
    echo "Bu bir shell script fonksiyonudur."
}

# Fonksiyonun çağrılması
fonksiyon_adi
```

→ **fonksiyon_adi()** fonksiyonu tanımlanmıştır.

→ Fonksiyonun içinde sadece bir **echo** komutu vardır.

→ Fonksiyon çağrıldığında, içinde tanımlanan komutlar çalışır ve **"Bu bir shell script fonksiyonudur."** yazısı ekrana basılır.

3. Fonksiyonlara Parametre Gönderme

Shell fonksiyonlarına argümanlar/parametreler gönderilebilir. Fonksiyona verilen parametreler, **$1, $2**, ... şeklinde değişkenler aracılığıyla alınır.

```
#!/bin/bash

# Parametre alan fonksiyon
selamla() {
    echo "Merhaba, $1!"
}

# Fonksiyonun çağrılması ve parametre verilmesi
selamla "Ahmet"
```

→ **selamla** fonksiyonu bir parametre alır (**$1**).

→ Fonksiyon çağrıldığında **"Ahmet"** parametresi verilmiştir ve çıktı **"Merhaba, Ahmet!"** olacaktır.

4. Birden Fazla Parametre ile Fonksiyon

Bir fonksiyona birden fazla parametre de verilebilir. Parametreler sırasıyla $1, $2, $3 gibi değişkenlerle fonksiyon içinde kullanılabilir.

```
#!/bin/bash

# İki parametre alan fonksiyon
topla() {
    toplam=$(( $1 + $2 ))
    echo "Toplam: $toplam"
}

# Fonksiyonun çağrılması ve iki parametre verilmesi
topla 5 10
```

→ **topla** fonksiyonu iki parametre alır.

→ Fonksiyon, iki sayıyı toplar ve sonucu ekrana yazdırır.

→ **topla 5 10** çağrıldığında, çıktı **"Toplam: 15"** olacaktır.

5. Fonksiyonlardan Değer Döndürme

Fonksiyonlar, shell script'te genellikle bir değer döndürmek için kullanılmazlar; bunun yerine çıktıyı **echo** ile döndürürler. Ancak **return** komutu kullanarak bir fonksiyonun çıkış durumunu (exit status) döndürebilirsiniz. Bu durum, hata denetimi yapmak için kullanışlıdır.

```
#!/bin/bash

# Değer döndüren fonksiyon
kontrol() {
    if [ $1 -gt 10 ]; then
        return 0   # Başarılı
    else
        return 1   # Başarısız
    fi
}

# Fonksiyonun çağrılması
kontrol 15

# Çıkış durumunun kontrolü
if [ $? -eq 0 ]; then
    echo "Sayı 10'dan büyük."
else
    echo "Sayı 10'dan küçük veya eşit."
fi
```

→ **kontrol** fonksiyonu, verilen sayının 10'dan büyük olup olmadığını kontrol eder.

→ **return 0** başarı anlamına gelir, **return 1** hata veya başarısızlık anlamına gelir.

→ **$?** ile fonksiyonun çıkış durumu kontrol edilir. Eğer **0** döndüyse, sayı 10'dan büyük demektir.

6. Yerel Değişkenler (Local Variables)

Shell fonksiyonlarında değişkenler varsayılan olarak küreseldir (global), yani fonksiyon dışındaki kod tarafından da erişilebilir. Ancak, bir değişkenin sadece fonksiyon içinde geçerli olmasını istiyorsanız, **local** anahtar kelimesini kullanarak yerel bir değişken tanımlayabilirsiniz.

```
#!/bin/bash

# Küresel değişken
isim="Ahmet"

# Yerel değişkeni olan fonksiyon
degistir_isim() {
    local isim="Ayşe"
    echo "Fonksiyon içinde isim: $isim"
}

# Fonksiyonun çağrılması
degistir_isim

# Fonksiyon dışındaki küresel değişken
echo "Fonksiyon dışında isim: $isim"
```

Çıktı:

```
Fonksiyon içinde isim: Ayşe
Fonksiyon dışında isim: Ahmet
```

→ Fonksiyon içinde **local isim="Ayşe"** olarak tanımlanan değişken sadece fonksiyon içinde geçerlidir.

→ Fonksiyon dışında aynı isimli bir değişken olsa da, fonksiyon dışına çıkıldığında küresel **isim** değişkeni değişmemiştir.

7.Fonksiyonlar ve return ile Hata Denetimi

Fonksiyonlar genellikle bir işlevi başarıyla tamamlayıp tamamlamadığını belirtmek için **return** komutunu kullanır. **return**, fonksiyonun durumunu belirtir ve genellikle **0** başarılı, **1** ve üzeri hatayı belirtir.

```
#!/bin/bash

# Dosya kontrol fonksiyonu
dosya_kontrol() {
    if [ -f "$1" ]; then
        return 0   # Dosya mevcut
    else
        return 1   # Dosya yok
    fi
}

# Fonksiyon çağrılması
dosya_kontrol "dosya.txt"

if [ $? -eq 0 ]; then
    echo "Dosya mevcut."
else
    echo "Dosya bulunamadı."
fi
```

→ **dosya_kontrol** fonksiyonu, verilen dosyanın mevcut olup olmadığını kontrol eder ve duruma göre 0 veya 1 döndürür.

→ **$?** ile fonksiyonun geri döndürdüğü değer kontrol edilir ve uygun mesaj gösterilir.

Shell Script'te Debugging (Hata Ayıklama)

Shell script'lerde yazılan kodlarda, hata ayıklama (debugging) işlemi önemlidir. Çünkü karmaşık veya uzun script'lerde hataların neden kaynaklandığını anlamak zor olabilir. Hataları bulmak ve gidermek için çeşitli debugging yöntemleri ve araçları kullanılır. Shell script debugging, script'in çalışma sırasında komutların nasıl çalıştığını ve sonuçlarını anlamaya yönelik teknikler içerir.

1. -x Parametresi ile Satır Satır İzleme

Shell script'lerde **-x** parametresi, script'in çalışırken her bir komutu ve çıktısını satır satır gösterir. Bu, hangi komutların çalıştığını ve ne sonuçlar döndüğünü görmek için faydalıdır.

Kullanımı:

```
bash -x script.sh
```

Veya script'in içinde **set -x** komutu eklenerek belirli bir bölümde debugging açılabilir.

```
#!/bin/bash

set -x  # Debugging başlar
echo "İlk komut"
echo "İkinci komut"
set +x  # Debugging biter

echo "Üçüncü komut"
```

Çıktı:

```
+ echo 'İlk komut'
İlk komut
+ echo 'İkinci komut'
İkinci komut
Üçüncü komut
```

set -x → Komutların çalışma sırasındaki detaylı çıktısını ekrana yazdırır.

set +x → Debugging'i durdurur ve komutlar normal şekilde çalışmaya devam eder.

2. -v Parametresi ile Komutların Görüntülenmesi

-v parametresi, script'in çalıştırdığı komutları ekrana yazdırır (ancak komutların çıktısını göstermez). Bu yöntem, komutların sırasıyla nasıl çalıştırıldığını görmek için kullanılır.

Kullanımı:

```
bash -v script.sh
```

Veya script içinde **set -v** komutu kullanılarak debugging açılabilir.

```
#!/bin/bash

set -v  # Komutları göster
echo "Merhaba dünya"
```

Çıktı:

```
 echo "Merhaba dünya"
Merhaba dünya
```

→ set -v: Script'in çalıştırdığı komutları yazdırır, ancak komutların sonuçlarını göstermez.

3. Hem -x Hem de -v Kullanımı

-x ve **-v** parametrelerini birlikte kullanarak hem komutları hem de bu komutların çıktısını aynı anda görüntüleyebilirsiniz.

```
bash -xv script.sh
```

→ **-v** komutları yazdırır.

→ **-x** ise komutların nasıl çalıştığını ve sonucunu gösterir.

Örnek1

```bash
#!/bin/bash

# Yedeklenecek dizin
hedef_dizin="/home/kullanici/dosyalar"

# Yedek dosyasının adını oluştur (tarih ile)
yedek_adi="yedek_$(date +%Y-%m-%d).tar.gz"

# Yedeğin kaydedileceği dizin
yedek_dizini="/home/kullanici/yedekler"

# Yedekleme işlemi
echo "Yedekleme başlıyor..."
tar -czf "$yedek_dizini/$yedek_adi" "$hedef_dizin"

# Başarı mesajı
if [ $? -eq 0 ]; then
    echo "Yedekleme başarıyla tamamlandı. Yedek dosyası: $yedek_dizini/$yedek_adi"
else
    echo "Yedekleme başarısız!"
fi
```

Bu script, belirli bir dizindeki dosyaları sıkıştırarak yedekler ve yedek dosyasını belirli bir dizine kaydeder.

→ **tar** komutunu kullanarak hedef dizini sıkıştırır.

→ Sıkıştırılmış dosyayı belirlenen **yedek_dizini** içine kaydeder.

→ Tarihe dayalı dosya adlarıyla yedekleme yapar.

Örnek2

```bash
#!/bin/bash

# Log dosyalarının bulunduğu dizin
log_dizini="/var/log/myapp"

# 30 günden eski log dosyalarını sil
find $log_dizini -type f -name "*.log" -mtime +30 -exec rm -f {} \;

# Sonuç mesajı
if [ $? -eq 0 ]; then
    echo "Eski log dosyaları başarıyla temizlendi."
else
    echo "Log temizleme işlemi başarısız!"
fi
```

Bu script, belirli bir dizindeki eski log dosyalarını kontrol eder ve 30 günden daha eski olanları siler.

→ **find** komutu ile 30 günden eski olan log dosyaları bulunur ve silinir.

→ Bu, sunucu üzerinde gereksiz disk kullanımı yaratan eski log dosyalarının düzenli olarak temizlenmesini sağlar.

Örnek3

```
#!/bin/bash

# Kullanıcı raporu oluşturma
echo "Kullanıcı Hesapları ve Son Giriş Zamanları" > user_report.txt
echo "----------------------------------------" >> user_report.txt

# Tüm kullanıcılar için son giriş zamanlarını göster
for user in $(cut -d: -f1 /etc/passwd)
do
    last_login=$(lastlog -u $user | tail -1 | awk '{print $4, $5, $6, $7}')
    echo "Kullanıcı: $user, Son Giriş: $last_login" >> user_report.txt
done

echo "Kullanıcı raporu oluşturuldu: user_report.txt"
```

Bu script, sistemdeki kullanıcı hesaplarını ve son giriş zamanlarını raporlar. Raporda, her kullanıcının son ne zaman giriş yaptığını gösterir.

→ Sistemdeki kullanıcıları **/etc/passwd** dosyasından okur.

→ Her kullanıcı için **lastlog** komutu ile son giriş zamanını alır ve raporlar.

→ Rapor, **user_report.txt** dosyasına kaydedilir.

Script'i Oluşturma Ve Çalıştırma

Öncelikle, Shell script'inizi bir dosyaya yazmanız gerekir. Bu dosya genellikle **.sh** uzantısıyla kaydedilir, ancak zorunlu değildir.

Bir script **oluşturmak** için aşağıdaki adımları izleyebilirsiniz:

→ Bir terminal açın.

→ Aşağıdaki komutla bir dosya oluşturun ve düzenlemeye başlayın:

```
nano script_adi.sh
```

→ Shell script kodunuzu bu dosyaya yazın. Örneğin:

```
#!/bin/bash
echo "Merhaba, bu bir shell script'tir!"
```

→ Kaydedip çıkın. (Nano editöründe `Ctrl + O` ile kaydedip, `Ctrl + X` ile çıkabilirsiniz.)

Bir script **çalıştırmak** için aşağıdaki adımları izleyebilirsiniz:

→ Bir Shell script'in çalıştırılabilmesi için dosyanın çalıştırılabilir olması gerekir. Bunu yapmak için **chmod** komutunu kullanarak dosyaya çalıştırma izni verebilirsiniz.

```
chmod +x script_adi.sh
```

Bu komut, **script_adi.sh** dosyasına çalıştırılabilirlik izni verir. Artık script'iniz çalıştırılabilir hale gelmiştir.

→ Script'i kendi yolunda çalıştırmak isterseniz, Terminalde script'in bulunduğu dizine gidin ve aşağıdaki komutu çalıştırın:

```
./script_adi.sh
```

Burada **./** dizindeki script'i çalıştırmak anlamına gelir.

→ Alternatif olarak, script'inizin çalıştırılabilir olup olmamasına bakmaksızın **bash** ya da **sh** komutuyla script'i çalıştırabilirsiniz:

```
bash script_adi.sh
```

veya

```
sh script_adi.sh
```

→ Eğer script'inizin uzun süre çalışacağını biliyorsanız ve terminali kullanmaya devam etmek istiyorsanız, script'i arka planda çalıştırabilirsiniz:

```
./script_adi.sh &
```

Bu komut, script'i arka planda çalıştırır ve terminalde çalışmaya devam etmenizi sağlar.