# KABUK PROGRAMLAMA

(shell programming- scripting)

# Kabuk Programlamaya Giriş

Her kabuğun kendine özgü programlama dili yapısı vardır. Bash kabuğu ise güçlü programlama özellikleriyle karmaşık programların rahatça yazılmasına izin verir. Mantıksal operatörler, döngüler , değişkenler ve modern programlama dillerinde bulunan pek çok özellik bash kabuğunda da vardır ve işleyiş tarzları da hemen hemen aynıdır.

Genellikle, bir programı oluşturacak olan komutlar bir dosyaya yazılırlar ve ardından bu dosya çalıştırılır. Herhangi bir editör yardımıyla yazılan program, daha sonra kabuk altında çalıştırılır. Bir kabuk programı diğerlerini çalıştırabilir. Bu düzende kabuk programlarını daha karmaşık komutların biraraya gelmiş ve yapısallaşmış haline benzetebiliriz.

Bash'in en büyük dezavantajı, derlenerek çalıştırılan dillere göre (C, C++ gibi) daha yavaş olması, sistem kaynaklarını biraz daha fazla tüketmesidir.

## *Kabuk Programları*

Kabuk programları, bir veya birden fazla Linux komutunu tutan dosyalardır. Bu dosya yaratıldıktan sonra doğrudan dosyanın ismi girilerek veya dosya isminden önce '.' karakteri getirerek çalıştırılabilir. Bir kabuk programı, çalıştırma bitini 1 yapmak suretiyle "çalıştırılabilir" hale getirilir. `chmod` komutu yardımıyla bir programı çalıştırılabilir yapmak için ,

```
$ chmod +x komut-ismi
```

yazılabilir. Bundan sonra programın ismi yazılıp enter tuşuna basıldığı zaman bir program Linux komutuymuş gibi çalışacaktır.

```
$ cat calistir
echo -n "Tarih : "
date
$ chmod +x calistir
$ calistir
Tarih : Sun Dec  8 07:11:51 EET 1996
```

Yukarıdaki örnekte "calistir" isimli iki satırlık bir kabuk programının önce içeriği ekrana yazıldı, ardından çalıştırılacak duruma getirildi ve çalıştırıldı.

Kabuk programları yazarken dosyanın işlevini ve her satırdaki komutun veya komut kümesinin ne amaçla kullanıldığını gösteren açıklama satırları kullanmak işe yarar. Bir açıklama eklemek için satır başına (veya boş satıra) # işareti eklenir ve ardından istenilen cümle girilir. # işaretinden sonraki tüm satır kabuk tarafından gözardı edilir. Aşağıdaki programda komut öncesinde yeralan açıklama satırı, komut hakkında bilgi veriyor.

```
# gunzip komutu dosya acmak icin kullanilir.
gunzip sistem.gz
```

Yorum satırı, komutun sonuna da eklenebilir.

```
ps -aux         # sistem surecleri hakkinda ayrintili bilgi..
```

Bir kabuk altında çalışırken başka bir kabuk için yazılmış bir programı çalıştırmak mümkündür. Örneğin tcsh altındasınız ve daha evvel bash kullanarak yazdığınız bir programı çalıştırmak istiyorsunuz. Önce bash yazarak kabuk değiştirmeli, ardından programı çalıştırmalı, ve tekrar tcsh'a dönmelisiniz. Tüm bunları otomatik olarak yaptırabilirsiniz. Programın en başına #! karakterini, ardından programın çalışacağı kabuğun patikasını yazın. Örneğin `#!/bin/bash` komutunu programın en üstüne eklerseniz bu program bash kabuğu altında çalışacaktır.

## *Değişkenlerin Kullanımı*

Bir değişkene değer atandığı anda sistem tarafından tanınır. Değişkenler alfabetik veya nümerik karakterlerden oluşabilirler fakat bir değişken sayısal bir değer ile başlayamaz. Bunların dışında değişken isminin içinde "_" karakteri de bulunabilir. Bir değişkene değer ataması "=" işareti yardımıyla yapılır.

```
$ mesaj="aksama yemege geliyorum"
```

İçeriği olan bir değişkene başına "$" işareti konularak ulaşılır. Aşağıda, `echo` komutu yardımıyla bir değişkenin içeriği ekrana basılıyor.

```
$ echo $mesaj
aksama yemege geliyorum
$ echo yarin $mesaj
yarin aksama yemege geliyorum
```

Aynı mesajı değişken kullanmadan da görüntüleyebiliriz.

```
$ echo "Aksama yemege geliyorum"
Aksama yemege geliyorum
```

## *Giriş/Çıkış İşlemleri*

Bir kabuk programı çalışırken kullanıcıdan klavye yardımıyla bilgi girmesi sağlanabilir. Bu tür işlemler için tanımlanan `read` komutu klavyeyi okur ve aldığı bilgiyi bir değişkene atar. Aşağıdaki komutları içeren program yardımıyla klavyeden okunan değer ekrana yazılıyor. **echo** komutundan sonra birden fazla değişken grubu veya hem değişken, hem de dizi kullanılabilir.

```
echo Bir sayi giriniz..
read sayi
echo Girilen sayi : $sayi
```

Bazı durumlarda girilen değer özel karakterleri içerebilir. Bu durumda istenmeyen bazı sonuçların doğması kaçınılmaz olur. Aşağıdaki örneği bir dosya içine yazın ve dosyayı çalıştırdıktan sonra "*" tuşuna basın.

```
echo Bir karakter giriniz
read a
echo Girdiginiz karakter : $a
```

**echo** komutundan gelecek bir yıldız işareti, bulunduğunuz dizindeki tüm dosyaları listeleyecektir.

## *Aritmetik İşlemler*

bash kabuğunda matematiksel işlemlere büyük sınırlamalar getirilmiştir. Tamsayı değişkeni dışında matematiksel değişken kullanmak için bu işlemler için geliştirilmiş ve kolaylıklar sağlayan **awk** veya **bc** kullanabilirsiniz.

Aritmetik işlemler için **eval** komutunu veya bash kabuğu altında yerleşik (builtin) komut olan `let` komutunu kullanabilirsiniz. Aşağıda let komutunun kullanımı görülüyor.

```
$ let "degisken=aritmetik islem"
```

Bu örnekte iki sayı çarpılıp çıkan sonuç başka bir değişkene yazılıyor.

```
$ let "carpim=2*7"
$ echo $carpim
```

Aritmetik değişken tanımlamanın diğer bir yolu da `typeset` komutu kullanmaktır.

```
$ typeset -i sonuc      (sonuc degiskeni bir dogal sayi icerecek)
$ a=100 ; b=56          (iki komutu ayirmak icin ; kullanilabilir)
$ sonuc=a*b
$ echo $sonuc
5600
```

Normal olarak bash, kesirli ve noktalı işlemleri yapamaz. Bunun için **bc** kullanabilirsiniz. **Bc**, çok yüksek duyarlılığa sahip bir hesap makinasıdır.

```
$ a=3.749
$ b=22.34
$ echo “$a*$b” | bc
83.752
```

## *if-else Kalıbı ve Kontrol İşlemleri*

Hemen her programlama dilinde olan `if` kalıbı bir Linux komutunun çalışmasını kontrol (`test`) eder. `if` komutu yerleşik bir komuttur. `if` komutunun ardından gelen Linux komutu çalıştırılır ve komutun çıkış durumu (exit status) gözönüne alınarak ardından gelen `then` deyimiyle birlikte devamı işletilir. Genellikle komutun iki türlü çıkış durumu olacağından `else` komutunun ardından gelen komut zinciri, diğer çıkış durumunda çalıştırılır. Her `if`, bir `fi` komutuyla bitmelidir. Aşağıda if-then-else komutunun örnek sözdizimi görülüyor.

```
if linux komutu
    then
        komut1
        komut2
        ...
    else
        komut1
        komut2
        ...
fi
```

`if` komutu genellikle kendine `test` komutu ile birlikte kullanım bulur. Bu komut yardımıyla mantıksal işlemler yapılabilir, sayılar ve hatta diziler karşılaştırılabilir. Anahtar sözcük olan `test`'ten sonra opsiyonlar ve/veya karşılaştırılacak olan değerler yazılır. Her opsiyon bir mantıksal işleme karşılık gelir. Örneğin `-lt` opsiyonu ilk girilen aritmetik değişkenin ikinci değerden küçük olup olmadığını denetler. Benzer şekilde = opsiyonu da iki karakter kümesinin eşitliğini kontrol eder. Aşağıda `test` komutunun örnek kullanımı yeralıyor.

```
$ test 5 -eq 3
$ a="linux"
$ test $a="linux"
```

komutun işletilmesinin ardından kabuğa bir değer döndürülür. Bu değer komut başarılı olarak işletilmişse 0, değilse 1'dir. Son çalıştırılan tüm Linux komutlarının çıkış değeri $? değişkeninde tutulur. test komutunun çıkış değeri de bu yolla öğrenilebilir.

```
$ sayi=4
$ test $sayi -eq 4
$ echo $?
0
$ test $sayi -lt 2
$ echo $?
1
```

`test` komutu yerine parantezler de kullanılabilir. Yukarıdaki iki örnek, parantez kullanılarak şu şekilde yazılabilir:

```
$ [ $sayi -eq 4 ]
$ [ $sayi -lt 12 ]
```

Dikkat edilmesi gereken bir nokta, köşeli parantez kullanırken araya boşlukların eklenmesidir. Parantezler başlı başına bir komut olarak görüldüklerinden sağında ve solunda en az bir boşluk bırakılmalıdır. `test` komutunda sıkça kullanılan diğer seçenekler şunlardır:

| Aritmetik karşılaştırma | |
|---|---|
| -gt | büyük |
| -lt | küçük |
| -ge | büyük eşit |
| -le | küçük eşit |
| -eq | eşit |
| -ne | eşit değil |
| Dizisel karşılaştırma | |
| -z | boş dizi |
| -n | tanımlı dizi |
| = | eşit diziler |
| != | farklı diziler |
| Dosya karşılaştırması | |
| -f | dosya var |
| -s | dosya boş değil |
| -r | dosya okunabilir |
| -w | dosyaya yazılabilir |
| -x | çalıştırılabilir dosya |
| -h | sembolik bağlantı |
| -c | karakter aygıt |
| -b | blok aygıt |
| Mantıksal karşılaştırma | |
| -a | VE |
| -o | VEYA |
| ! | DEĞİL |

`if` komutunun `test` ile birlikte kullanılabildiğini daha önce belirtmiştik. Aşağıda bununla ilgili küçük bir örnek yeralıyor.

```
#!/bin/bash
echo "0 ile 20 arasinda bir sayi secin"
read sec
if [ $sec -lt 10 ]
   then
        echo "Secilen sayi tek basamakli"
   else
        echo "Secilen sayi cift basamakli"
fi
```

Her `if` komutu bir `fi` ile son bulmalıdır.

## *case Kalıbı*

Birkaç alternatif arasından seçim yapmak için kullanılan bir komut olan `case`, bir eşleştirme gördüğü anda belirli bir komut kümesini işleme sokar. case yapısı `case` komutu ile başlar, eşleştirilecek olan anahtar sözcük yazılır ve seçenekler alt alta, her seçeneğe ait olan komutlarla birlikte belirtilir. Tüm yapı `esac` komutu ile son bulur.

```
case anahtar-sozcuk in
        secenek1)
                komutlar
                ;;
        secenek2)
                komutlar
                ;;
        *)
                komutlar
                ;;
esac
```

Seçenekler arasında özel karakterler *(*, [, ], ? gibi)* kullanılabilir. Hiçbir eşleme yapılmadığı zaman *) seçeneği değerlendirilecek ve buna bağlı olan komutlar işletilecektir. * kullanımı isteğe bağlıdır. Aşağıda `case` komutuna ilişkin kısa bir örnek veriliyor.

```
#!/bin/bash

clear
echo    "1. ekrani temizle"
echo    "2. sistemdekileri goruntule"
echo    "3. dizindeki dosyalari goster"

echo -n "Secenegi giriniz : "
read secenek

case $secenek in
```

```
        1)
          clear
          ;;
        2)
          w
          ;;
        3)
          ls -al
          ;;
        *)
          echo Hatali secenek
esac
```

# Döngüler

Diğer hemen tüm programlama dillerinin en büyük gücü olan döngü işlemlerine kabuk altında da izin veriliyor. Burada programcı tarafından en çok kullanılan 2 döngü tipi anlatılacaktır: `while` ve `for`. `while` komutu her döngüde bir denetleme mekanizmasını harekete geçirirken `for` döngüsü bir listenin elemanlarını sırayla seçer.

## *while-do Döngüsü*

Döngü bloğu `while` anahtar kelimesiyle başlar, ardından gelen koşul sağlandığı sürece döngü işletilir. Önce koşulun sağlanıp sağlanmadığına bakılır. Döngüden çıkabilmek için mutlaka döngü içindeki koşul ifadesinin değerini yanlış yapacak bir durum oluşmalıdır, aksi halde sonsuz döngü oluşur.

```
while kosul ifadesi
   do
        komutlar
   done
```

`if` komutuyla birlikte kullanılan `test` komutu, `while` döngüsünde koşul ifadesi olarak da yer alabilir. Aşağıda 1'den 100'e kadar sayan ve ekrana basan bir döngü görülüyor.

```
#!/bin/bash
deger=0
while [ $deger -lt 100 ]
   do
        deger=$((deger+1))
        echo $deger
   done
```

Yukarıda kullanılan (( ve )) karakterleri arasına matematiksel bir işlem getirilebilir. Bu özellik bash kabuğuna özgüdür.

## *for-do döngüsü*

Bir liste dahilindeki tüm değerlere sırayla erişimi sağlar. for komutundan sonra yeralan liste sırayla kullanılır ve herbirisi için döngü çalıştırılır. Listenin sonuna gelindiğinde ise döngüden çıkılır.

```
for degisken1 in deger1 deger2 ... degerX
   do
        komutlar
   done
```

Aşağıdaki örnek bu döngüyü kullanarak ekrana bir dizi kelime yazıyor. Döngü boyunca akasya, elma ve visne kelimeleri "agac" değişkenine kopyalanıyor ve her döngüde bu değişkenin içerdiği bilgiler ekrana yazılıyor.

```
for agac in akasya elma visne
   do
        echo $agac
   done
```

for-do döngüsü, dosya isimleri üzerinde yapılan işlemlerde de büyük kolaylıklar sağlar. Bunun için özel karakterlerden yararlanmak da olasıdır. Örnek olarak * karakteri o anki çalışma dizini içindeki tüm dosyaları seçer.

```
for a in * ; do
        file $a
        done
```

# Örnek Kabuk Programı

Sistem görevlisinin en çok kullandığım komutlardan birisi

```
ps -aux grep -i xxx
```

komutudur. Bu satır yardımıyla çalışan *xxx* isimli program hakkında daha detaylı bilgi elde edilebilir. `ps` komutu detaylı bir süreç listesini ekrana verirken çıktı doğrudan grep komutuna yönlendirilir ve sadece istediğimiz bilgi ekranda görünür. Fakat her zaman aynı uzun satırı tekrar tekrar yazmaktansa bu satırı bir dosyaya gönderip, dosya adını komut satırından çalıştırmak zamandan tasarruf sağlayacaktır. Biz de öyle yapalım ve aşağıdaki satırları ``goster'' isimli dosyaya yazalım.

```
#!/bin/bash
if [ $# = 1 ]
  then
  ps -ax | grep -i $1
else
  ps -ax
fi
```

Ardından dosyayı PATH değişkeninin işaret ettiği dizinlerden

```
/usr/local/bin
```

altına yerleştirip çalıştırılabilir olması için

```
chmod  +x /usr/local/bin/goster
```

komutunu uygulayın. Kendi yazdığınız dosyaları

```
/usr/local
```

veya

```
~/bin
```

dizini altına kopyalamanız bunların derli toplu olarak tek bir dizinde her an erişilebilir şekilde durmaları açısından önem taşır.

Dosyaya biraz daha yakından bakalım. İlk satır, bu dosyanın `/bin/bash` programı tarafından çalıştırılacağını gösterir. İkinci satır yardımıyla komut satırı üzerinde kaç tane opsiyon olduğu bulunur. $# çevresel değişkeni her biri TAB veya boşluk karakteri ile ayrılmış komut satırı opsiyonları sayısını verir. Aşağıdaki komutta toplam 3 opsiyon vardır.

```
$ ls --8bit  -F -b
```

`$#` komutuna benzer şekilde $1, $2, $3 ... değişkenleri de opsiyonları verir. Yukarıdaki komutta $1, $2, $2 değişkenleri sırasıyla

```
--8bit
-F
-b
```

değerlerini alırlar. Komut satırından çalıştırılan komut ise $0 değişkenine atılır. Yukarıdaki örnekte $0 değişkeni `ls` değerini tutacaktır.

`goster` dosyasına tek opsiyon yollayacağız. Bu opsiyon da hakkında detaylı bilgi alacağımız süreç olacaktır. Dosya içinde $1 değişkenini `ps` komutuna yerleştirmek için :

```
ps -ax | grep -i $1;
```

yazılır. Bundan sonra komut satırında, örneğin:

```
$ goster bash
```

girilirse $1 değişkeni `bash` e eşit olacak ve bu da

```
ps -ax grep -i bash
```

eşdeğer komutunu çalıştıracaktır. Komut olarak sadece `goster` girilirse $# değişkeni 0'a eşit olacağından dosyada yeralan ve hiç bir filtreleme yapmayan

```
ps -ax
```

komutu çalışır.