6. Pętla for

```
for i in {1..5}; do
echo "Wypisz $i"
done
for i in {5..50..5}; do
echo "Wypisz $i"
done
```

Wynik:

```
Wypisz 1
Wypisz 2
Wypisz 3
Wypisz 4
Wypisz 5
Wypisz 5
Wypisz 10
Wypisz 15
Wypisz 20
Wypisz 25
Wypisz 30
Wypisz 35
Wypisz 40
Wypisz 45
Wypisz 50
```

Wytłumaczenie:

Pętla for różni się składniowo od większości języków programowania.

W drugim przykładzie: pierwsza element po in to początkowa wartość iteratora, środkowa to limit kończący pętle, a trzecia operacja na iteratorze po każdym przejściu pętli.

7. Pętla while

```
x=1;
while [ $x -le 10 ]; do
echo "Iteracja: $x"
x=$[x + 1]
done
```

Wynik:

```
Iteracja: 1
Iteracja: 2
Iteracja: 3
Iteracja: 4
Iteracja: 5
Iteracja: 6
Iteracja: 7
Iteracja: 8
Iteracja: 9
Iteracja: 10
```

Wytłumaczenie:
Przykład pętli while analogiczny do większości języków programowania.
Operator –le w instrukcji warunkowej oznacza less than or eqal to (mniejsze bądź równe)

8. **Pętla while** (tu: wypisanie zawartości pliku)

```
cat file.txt | while read line; do
echo $line
done
```

Wynik zależy od zawartości pliku file.txt w tym samym katalogu).

Wytłumaczenie:
Pętle while możemy stosować także na takich obiektach jak pliki. Wyrażenie read line przyjmie wartość false, gdy w pliku nie będzie już więcej linii do przeiterowania => tym samym zakończy się pętla.

9. Komentarze

```
# komentarz w jednej linii
: '
komentarz
wielo
liniowy
'
```

Analogicznie jak w większości języków programowania.

10. Case

```
echo "Jesteś kobietą?"
read a
case $a in

[Tt]ak|[Pp]rawda)
echo "Odpowiedziałeś tak/prawda"
;;
[Nn]ie|[Ff]ałsz)
echo "Odpowiedziałeś nie/fałsz"
;;
esac
```

Wynik:

```
Jesteś kobietą?

tak  # odpowiedź użytkownika

Odpowiedziałeś tak/prawda

testy@vps515609:~/bashowanie$  ./current.sh  # ponowne uruchomienie
skryptu

Jesteś kobietą?

nie

Odpowiedziałeś nie/fałsz
```

Wytłumaczenie:

Konstrukcja case wykorzystywana jest dla rozbudowanych instrukcji warunkowych.
Operacja logiczna: [Tt]ak|[Pp]rawda po rozwinięciu oznacza:
Tak lub tak lub Prawda lub prawda

Porównywanie plików, łańcuchów i liczb

W interakcyjnych programach pojawia się wiele pytań i akcja programu zależy od odpowiedzi na nie. Do sprawdzania odpowiedzi służy polecenie test. Wynikiem polecenia test jest albo „0” (prawda) albo „1” (fałsz). Wynik ten jest przechowywany w zmiennej „$?”.

Polecenie test ma bardzo dużo parametrów.

Parametry dotyczące plików

-d cel sprawdza, czy cel jest istniejącym katalogiem:

```
#!/bin/bash
a=/var
b=/etc/profile
```

```
test -d $a
echo $a: $?
test -d $b
echo $b: $?
```

Poniżej podano wynik wykonania polecenia :

-e cel sprawdza, czy cel jest istniejącym plikiem:

```
#!/bin/bash
a=/var
b=2
test -e $a
echo $a: $?
test -e $b
echo $b: $?
```

Wynikiem może być:

```
/var: 0
2: 1
```

-f cel sprawdza, czy cel istnieje i jest normalnym plikiem:

```
#!/bin/bash
a=/var
b=/etc/profile
test -f $a
echo $a: $?
test -f $b
echo $b: $?
```

Wynik wygląda mniej więcej tak:

```
/var: 1
/etc/profile: 0
```

-r cel, -w cel, -x cel sprawdza, czy użytkownik ma określone uprawnienia: odczytu (-r), zapisu (-w), wykonania (-x)

```
#!/bin/bash
a=/etc/profile
test -r $a
echo "Read: $?"
test -w $a
echo "Write: $?"
test -x $a
echo "Execute: $?"
```

Plik /etc/profile ma następujące uprawnienia: rw-r—r-- . Użytkownik, który uruchomił ten skrypt może tylko czytać ten plik.

Rezultat polecenia będzie miał postać:

```
Read: 0
Write: 1
Execute: 1
```

-s cel sprawdza, czy plik jest pusty

```
#!/bin/bash
a=/etc/profile
test -s $a
echo $?
```

Rezultatem wykonania polecenia test powinno być `0`.

plik1 -nt plik2 sprawdza, czy plik plik1 jest nowszy od plik2

Porównuje daty ostatniej modyfikacji, a nie utworzenia

```
#!/bin/bash
a=/etc/profile
b=/var/log/Xorg.0.log
test $a -nt $b
echo "$a -nt $b:" $?
```

Wynik powinien być równy „1” (fałsz):

```
/etc/profile -nt /var/log/Xorg.0.log: 1
```