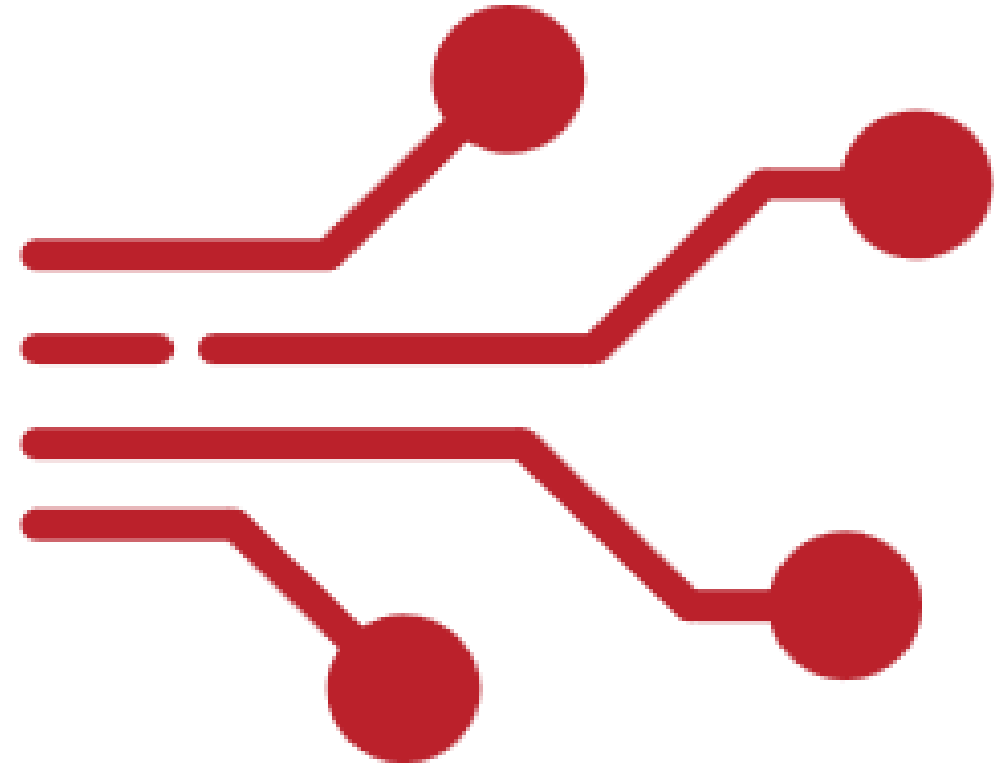




Programowanie skryptowe

dr Rafał Stęgierski,
*Politechnika Lubelska, Katedra Inteligencji
Obliczeniowej*

Charakterystyka języków programowania



Języki Kompilowane

- Kod źródłowy jest tłumaczony na kod maszynowy przed uruchomieniem programu przez kompilator.
- Kompilacja generuje plik wykonywalny.
- Kod jest zoptymalizowany dla architektury docelowej.
- Program działa bezpośrednio na procesorze.

Proces kompilacji

- Kod źródłowy jest tłumaczony do kodu maszynowego.
- Kod maszynowy zostaje poddany optymalizacji.
- Konsolidator łączy kod maszynowy z wywołaniami bibliotecznymi.
- Wynik zostaje zachowany w formie binarnej zawierającej bezpośrednie rozkazy procesora.

Przykłady Języków Kompilowanych

- C/C++:
 - Niskopoziomowe, bardzo szybki kod wynikowy.
- Rust:
 - Bezpieczny, nowoczesny, systemy o wysokiej wydajności.
- Go:
 - Szybki, prosty, wydajne aplikacje sieciowe i mikroserwisy.

Proces interpretacji

- Kod jest wykonywany linia po linii przez interpreter w trakcie działania programu, bez wcześniejszej kompilacji.
- Brak etapu kompilacji.
- Interpreter tłumaczy kod w czasie rzeczywistym.
- Łatwiejsze debugowanie i testowanie.

Przykłady języków interpretowanych

- Python:
 - Wszechstronny, szybki rozwój.
- JavaScript:
 - Dynamiczny, aplikacje webowe.
- Ruby:
 - Elegancki, produktywny, aplikacje webowe.

Wydajność

- Kompilowane znacznie szybsze, bo kod maszynowy jest wcześniej przygotowany i jest tylko przekazywany do wykonywania bezpośrednio przez procesor.
- Interpretowane są wolniejsze, bo proces tłumaczenia do kodu maszynowego ma miejsce w czasie wykonania programu.

Zalety języków kompilowanych

- Wysoka wydajność, bo kod jest zoptymalizowany.
- Większe bezpieczeństwo, między innymi trudniejsze ataki typu buffer overflow.
- Wczesne wykrywanie błędów podczas kompilacji.
- Brak zależności od istnienia interpretera w systemie, ze względu na niezależny plik wykonywalny.

Wady języków kompilowanych

- Wolniejszy rozwój aplikacji, bo etap kompilacji zabiera czas.
- Zależność od platformy, ze względu na różny kod maszynowy i format pliku wykonywalnego.
- Trudniejsze debugowanie, często trzeba rekompilować kod po zmianach.
- Mniejsza elastyczność trudniejsze modyfikowanie dynamiczne.

Zalety języków interpretowanych

- Szybki rozwój aplikacji dzięki uruchomieniu natychmiast po edycji.
- Przenośność oznaczająca ten sam kod na różnych platformach.
- Elastyczność, zapewniana przez dynamiczne typowanie, czy modyfikacje w locie.
- Łatwiejsze testowanie za sprawą błędów wykrywane podczas wykonania z możliwością modyfikacji w locie.

Wady języków interpretowanych

- Niższa wydajność kodu oznaczająca jego wolniejsze wykonanie.
- Zależność od środowiska uruchomieniowego, a przynajmniej interpretera.
- Brak zaawansowanej optymalizacji w porównaniu do wariantu kompilowanego.

Podejście hybrydowe JIT

- Just-In-Time Compilation łączy zalety obu podejść.
- Kompilacja w trakcie wykonania programu.
- Kompilacja do kodu pośredniego.
- Optymalizacja w czasie działania oznaczająca lepsza wydajność.
- Ostatecznie oznacza przenośność i szybkość wykonania.

Działanie JIT

- Krok 1: Kod kompilowany do bytecode (kod pośredni).
- Krok 2: Bytecode wykonywany przez maszynę wirtualną.
- Krok 3: JIT kompiluje "gorące" fragmenty do kodu maszynowego.
- Efekt: Szybkość aplikacji wynikowej bliska językom kompilowanym.

Przykłady języków JIT

- Java (HotSpot):
 - JVM kompiluje bytecode do natywnego kodu maszynowego.
- C# (.NET):
 - CLR używa JIT do kompilacji IL w czasie uruchomienia.
- JavaScript (V8):
 - TurboFan kompiluje gorący kod dla maksymalnej wydajności.

Zalety kompilacji JIT

- Optymalizacja adaptacyjna i dostosowanie do rzeczywistego użycia.
- Przenośność, bo bytecode działa na różnych platformach.
- Lepsza wydajność niż czysty interpreter.
- Profile-guided optimization jako optymalizacja na podstawie danych.

Wady kompilacji JIT

- Warm-up time, czyli czas, który środowisko potrzebuje na wystartowanie.
- Większe zużycie pamięci, ze względu na kompilator, bytecode i wynikowy kod maszynowy.
- Nieprzewidywalna wydajność, zawsze pierwsza iteracja wolniejsza ze względu na czas kompilacji.
- Złożoność wynikająca z konieczności implementacji maszyny wirtualnej.

17

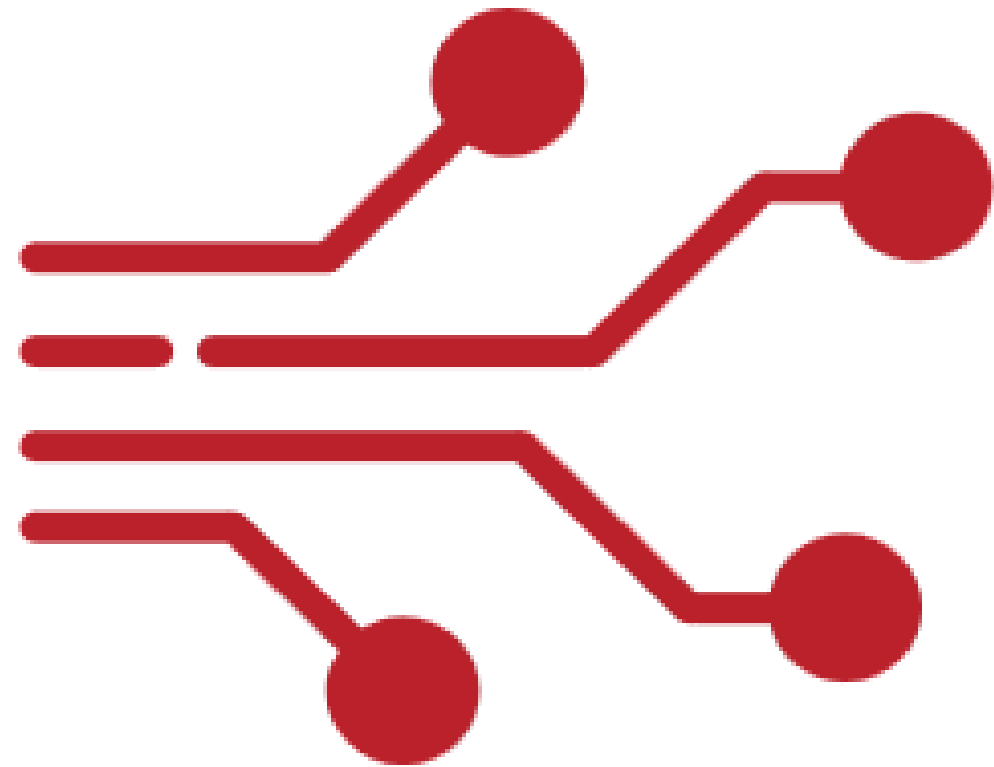
Kiedy język kompilowany?

- Aplikacje wymagające maksymalnej wydajności.
- Systemy operacyjne i sterowniki.
- Gry komputerowe.
- Aplikacje dla urządzeń obsadzonych.

Kiedy język interpretowany?

- Prototypowanie i szybki rozwój.
- Aplikacje webowe i skrypty.
- Analiza danych i machine learning.
- Projekty wymagające dużej elastyczności.

Skrypty powłoki



20

Czym są skrypty powłoki

- Języki programowania służące do automatyzacji zadań w systemie operacyjnym poprzez interfejs wiersza poleceń.
- Interpretowane.
- Zintegrowane z systemem operacyjnym.
- Upraszczają zarządzanie plikami, procesami i usługami.

Bash (Bourne Again Shell)

- Najpopularniejsza powłoka w systemach Linux i macOS.
- Domyślna w większości dystrybucji.
- Standard w środowiskach Unix/Linux.
- Ogromna baza skryptów i dokumentacji.
- Wsparcie dla potoków, przekierowań, zmiennych.

PowerShell

- Nowoczesna powłoka Microsoft oparta na .NET.
- Działa na Windows, Linux i macOS.
- Obiektowa, co oznacza operacje na obiektach, nie tekście.
- Wprowadza cmdlets jako polecenia ze spójną składnią.
- Mocno zintegrowana z ekosystemem Microsoft.

Zsh (Z Shell)

- Rozszerzona wersja Bash z dodatkowymi funkcjami.
- Domyślna powłoka w macOS od 2019.
- Zaawansowane autouzupełnianie.
- Konfigurowalność, w tym motywy i pluginy (Oh My Zsh).
- Kompatybilność wsteczna z Bash.

Fish (Friendly Interactive Shell)

- Nowoczesna powłoka z naciskiem na łatwość użycia i inteligentne podpowiedzi.
- Automatyczne sugestie na podstawie historii.
- Kolorowanie składni out-of-the-box.
- Prostsza składnia niż Bash/Zsh.

Skrypt Bash

```
#!/usr/bin/bash

THRESHOLD_BIG=$((1024 * 1024))      # 1 MB
THRESHOLD_MED=$((100 * 1024))      # 100 KB

for file in *; do
    if [[ -f "$file" ]]; then
        size=$(stat -c%s "$file")

        if (( size > THRESHOLD_BIG )); then
            echo "DUŻY : $file ($size B)"
        elif (( size > THRESHOLD_MED )); then
            echo "ŚREDNI: $file ($size B)"
        else
            echo "MAŁY : $file ($size B)"
        fi
    fi
done
```

26

Skrypt PowerShell

```
param(  
    [int64]$ThresholdBig = 1MB,  
    [int64]$ThresholdMed = 100KB  
)  
  
Get-ChildItem -File | ForEach-Object {  
    $size = $_.Length  
  
    if ($size -gt $ThresholdBig) {  
        Write-Host "DUŻY : $($_.Name) ($size B)"  
    }  
    elseif ($size -gt $ThresholdMed) {  
        Write-Host "ŚREDNI: $($_.Name) ($size B)"  
    }  
    else {  
        Write-Host "MAŁY : $($_.Name) ($size B)"  
    }  
}
```

27

Skrypt Fish

```
#!/usr/bin/fish

set THRESHOLD_BIG (math "1024 * 1024") # 1 MB
set THRESHOLD_MED (math "100 * 1024") # 100 KB

for file in *
    if test -f "$file"
        set size (stat -c '%s' "$file")

        if test $size -gt $THRESHOLD_BIG
            echo "DUŻY : $file ($size B)"
        else if test $size -gt $THRESHOLD_MED
            echo "ŚREDNI: $file ($size B)"
        else
            echo "MAŁY : $file ($size B)"
        end
    end
end
```

28

Zalety skryptów powłoki

- Automatyzacja: powtarzalne zadania wykonywane automatycznie.
- Szybkość uruchomienia: natychmiastowe wykonanie bez kompilacji.
- Integracja: bezpośredni dostęp do poleceń systemowych.
- Przenośność: skrypty działają na różnych maszynach z odpowiednim interpreterem.

Wady skryptów powłoki

- Ograniczona funkcjonalność: trudne algorytmy trudne do implementacji.
- Bezpieczeństwo: łatwo popełnić błędy z uprawnieniami.
- Debugowanie: trudniejsze niż w pełnych środowiskach rozwojowych z dedykowanymi narzędziami.
- Kompatybilność: różnice między powłokami/systemami.

30

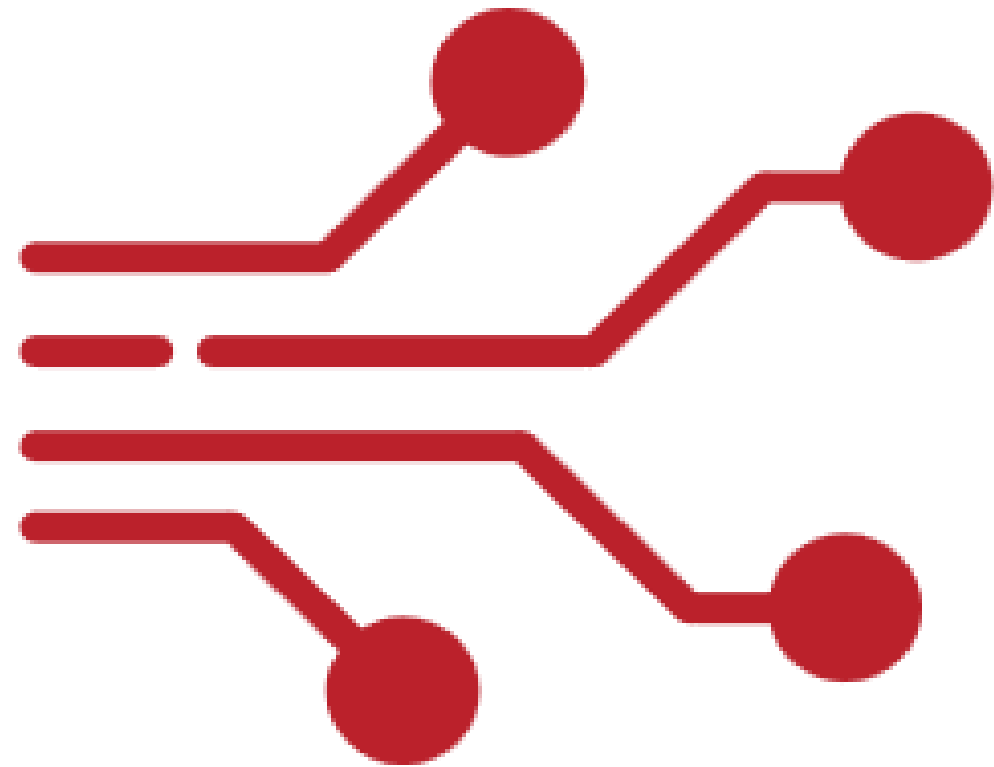
Standardowe zastosowania

- DevOps i CI/CD: automatyzacja budowania i wdrożeń aplikacji.
- Administracja systemem: backupy, monitoring, konfiguracja.
- Przetwarzanie plików: masowe operacje na plikach.
- Uruchamianie aplikacji: konfiguracja środowiska, zarządzanie wariantami uruchomienia.

Before we start... good practice ;-)

- Używaj shebang, czyli określ interpreter na początku.
- Sprawdzaj kody błędów, uwzględnij obsługę wyjątków.
- Cytuj zmienne i zapobiegaj problemom z białymi znakami.
- Komentuj kod i ułatw innym, i sobie, zrozumienie skryptu.

Środowiska pracy ze skryptami powłoki



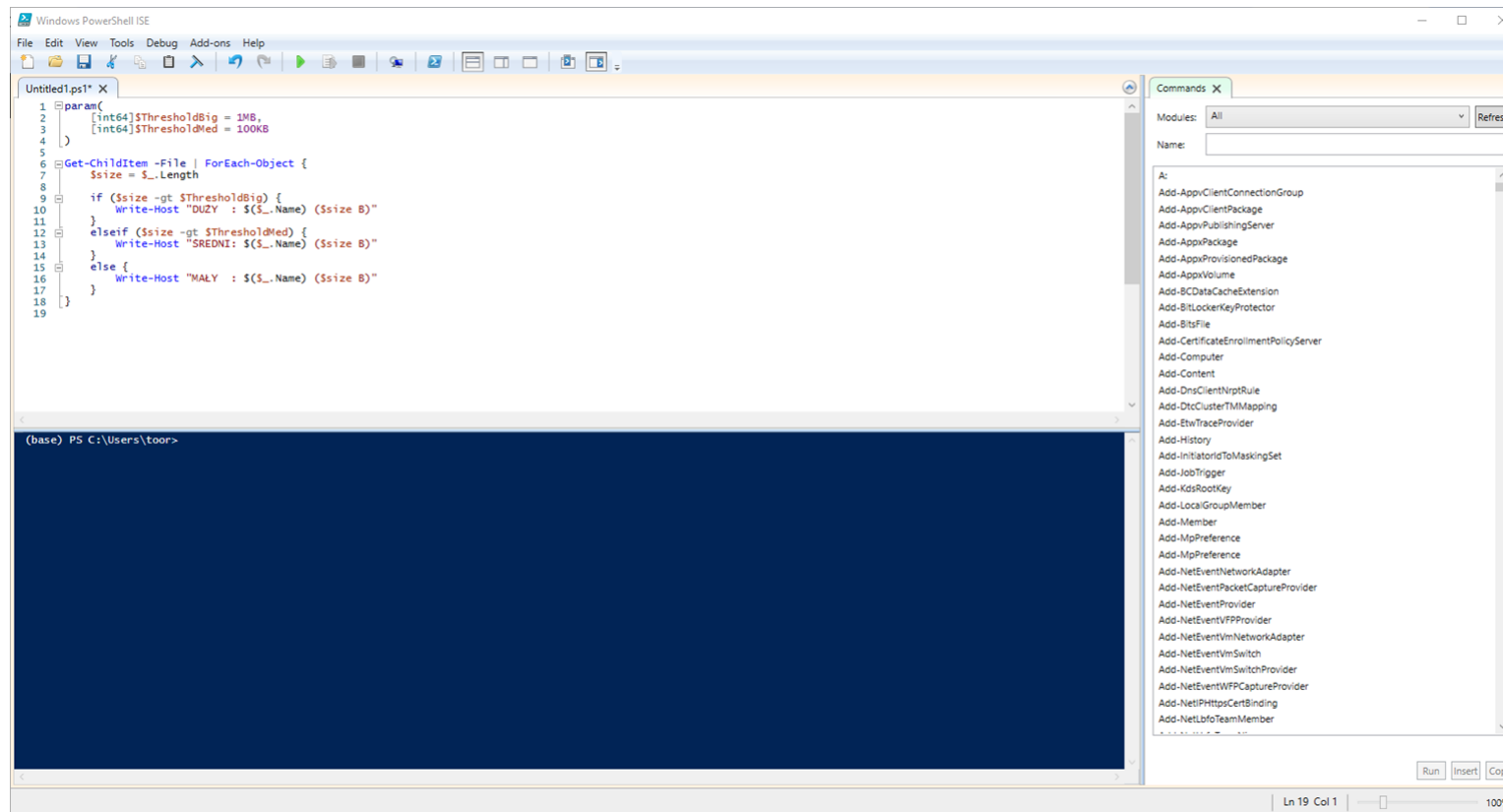
Absolutne minimum - edytor tekstu

- nano/pico: prosty edytor terminala, łatwy dla początkujących.
- vim/vi: złoty standard, zaawansowany edytor terminala.
- emacs: potężny, konfigurowalny edytor z rozbudowanymi funkcjami.
- Notepad: w sumie... czemu nie?
- Notepad++ (Windows): lekki edytor z kolorowaniem składni.

IDE i Zaawansowane Edytory

- Visual Studio Code/VSCodium: przypuszczalnie najpopularniejszy, rozszerzenia dla Bash/PowerShell, debugger.
- Geany lekkie, wieloplatformowe środowisko rozwojowe.
- PowerShell ISE: dedykowane IDE dla PowerShell (Windows).

PowerShell ISE



```
1 param(  
2     [int64]$ThresholdBig = 1MB,  
3     [int64]$ThresholdMed = 100KB  
4 )  
5  
6 Get-ChildItem -File | ForEach-Object {  
7     $size = $_.Length  
8  
9     if ($size -gt $ThresholdBig) {  
10        Write-Host "DUZY : $($_.Name) ($size B)"  
11    }  
12    elseif ($size -gt $ThresholdMed) {  
13        Write-Host "SREDNI: $($_.Name) ($size B)"  
14    }  
15    else {  
16        Write-Host "MALY : $($_.Name) ($size B)"  
17    }  
18 }  
19
```

Commands X

Modules: All Refresh

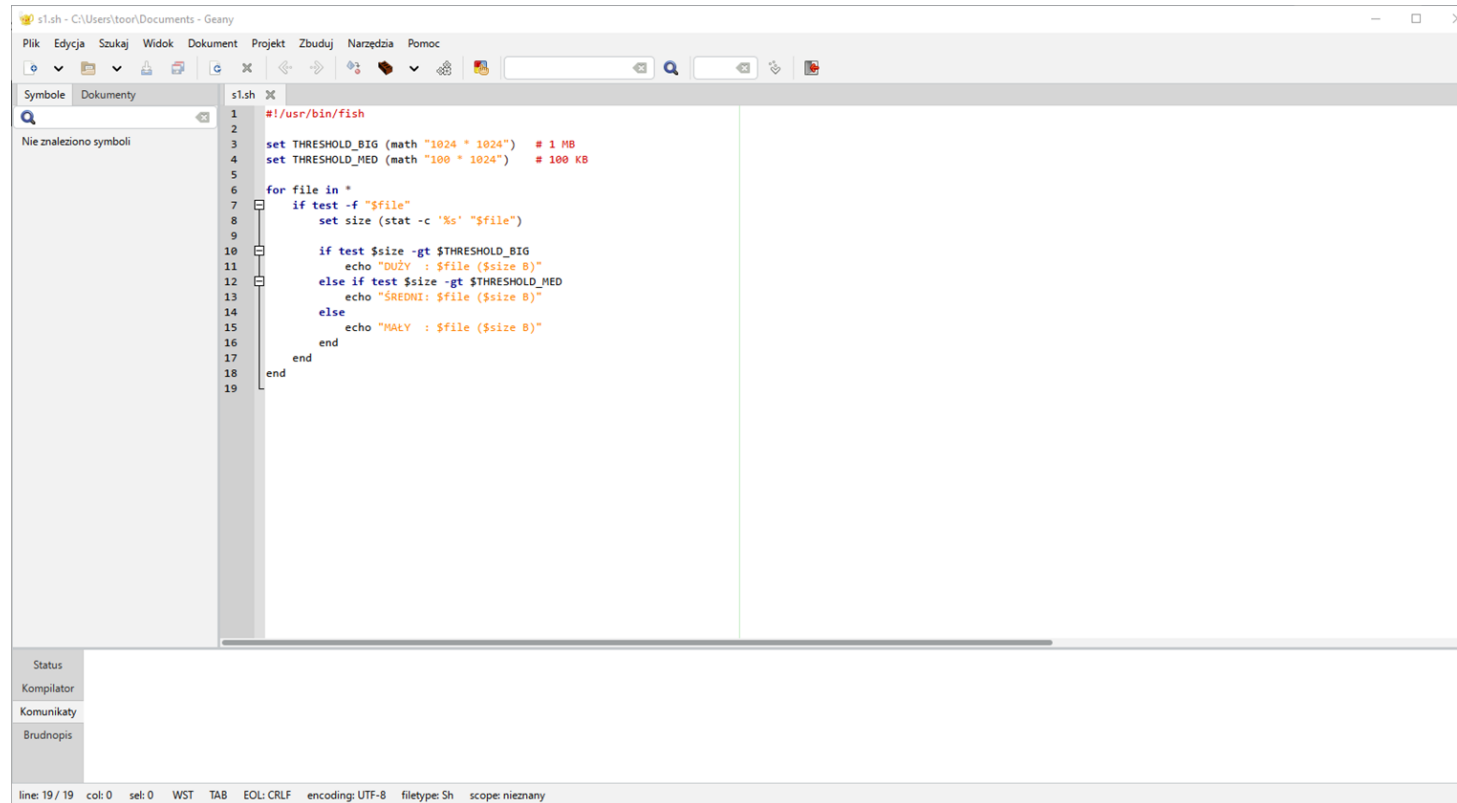
Name:

- Add-AppvClientConnectionGroup
- Add-AppvClientPackage
- Add-AppvPublishingServer
- Add-AppxPackage
- Add-AppxProvisionedPackage
- Add-AppxVolume
- Add-BCDataCacheExtension
- Add-BitLockerKeyProtector
- Add-BitsFile
- Add-CertificateEnrollmentPolicyServer
- Add-Computer
- Add-Content
- Add-DnsClientNrptRule
- Add-DtcClusterTmmMapping
- Add-EtwTraceProvider
- Add-History
- Add-InitiatorIdToMaskingSet
- Add-JobTrigger
- Add-KdsRootKey
- Add-LocalGroupMember
- Add-Member
- Add-MpPreference
- Add-MpPreference
- Add-NetEventNetworkAdapter
- Add-NetEventPacketCaptureProvider
- Add-NetEventProvider
- Add-NetEventVFPProvider
- Add-NetEventVmNetworkAdapter
- Add-NetEventVmSwitch
- Add-NetEventVmSwitchProvider
- Add-NetEventWFPProvider
- Add-NetEventWFPProvider
- Add-NetIcmpv4CertBinding
- Add-NetLbfoTeamMember

(base) PS C:\Users\toor>

36

Geany



```
1 #!/usr/bin/fish
2
3 set THRESHOLD_BIG (math "1024 * 1024") # 1 MB
4 set THRESHOLD_MED (math "100 * 1024") # 100 KB
5
6 for file in *
7   if test -f "$file"
8     set size (stat -c '%s' "$file")
9
10    if test $size -gt $THRESHOLD_BIG
11      echo "DUŻY : $file ($size B)"
12    else if test $size -gt $THRESHOLD_MED
13      echo "ŚREDNI: $file ($size B)"
14    else
15      echo "MAŁY : $file ($size B)"
16    end
17  end
18 end
19
```

37

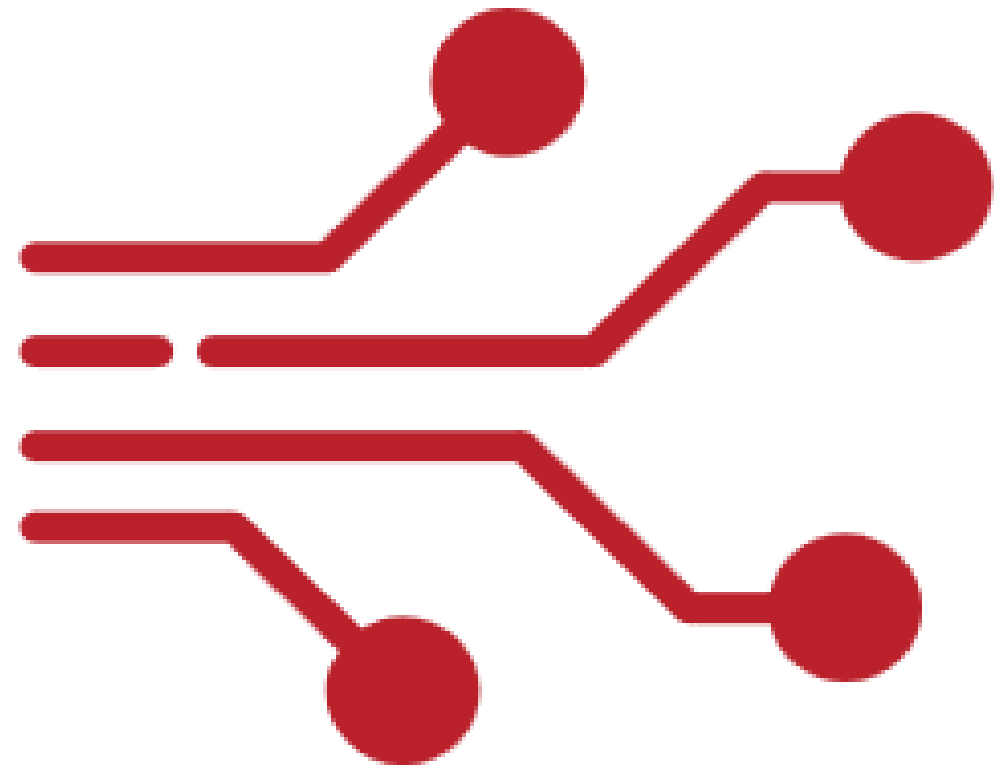
Kluczowy zestaw funkcji edytora

- Kolorowanie składni dla czytelności.
- Autouzupełnianie, w tym sugestie poleceń i zmiennych.
- Linting pozwalający na szybkie wykrywanie błędów przed uruchomieniem.
- Snippety, czyli szablony często używanych konstrukcji.
- Opcjonalnie wsparcie AI.

Uruchamianie skryptów

- Bezpośrednio: **./script.sh** (wymaga uprawnień +x) i shebang (np.: **#!/bin/bash**) na początku pliku.
- Przez interpreter: **bash script.sh**, **pwsh script.ps1**.
- Source/dot – **. script.sh** (wykonanie w bieżącej powłoce).

Drobne przypomnienie



40

Szablony

- **?** - dowolny znak.
- ***** - dowolny ciąg znaków.
- **[abc]** - dowolny znak ze zbioru.
- **[!abc]** - dowolny znak spoza zbioru.
- **[a-c]** - dowolny znak z zakresu.
- **[a-z]** - dowolna mała litera.
- **[A-Z]** - dowolny wielka litera.
- **[!0-9]** - dowolny znak poza cyframi.
- **[a-zA-Z]** - wszystkie litery.
- **[a-zA-Z_-]** - wszystkie litery, podkreślenie i myślnik.

41

Znaki specjalne 1

- ~ - katalog domowy.
- # - komentarz.
- \$ - wyrażenie zmienne.
- (- otwarcie podpowłoki.
-) - zamknięcie podpowłoki.
- \ - znak ucieczki.
- (- potok.
- { - otwarcie bloku poleceń.
- } - zamknięcie bloku poleceń.

42

Znaki specjalne 2

- ; - separator poleceń.
- ' - łańcuch nie-interpretowany.
- ” - łańcuch (częściowo) interpretowany.
- < - przekierowanie wejście.
- > - przekierowanie wyjścia.

Przykład użycia znaków specjalnych

```
toor@Mikasa:~$ echo oczywistością jest, że 3 * 3 > 5 jest prawdziwe
toor@Mikasa:~$ echo 'oczywistością jest, że 3 * 3 > 5 jest prawdziwe'
oczywistością jest, że 3 * 3 > 5 jest prawdziwe
toor@Mikasa:~$ echo oczywistością jest, że '3 * 3 > 5' jest prawdziwe
oczywistością jest, że 3 * 3 > 5 jest prawdziwe
toor@Mikasa:~$ echo oczywistością jest, że 3 \* 3 \> 5 jest prawdziwe
oczywistością jest, że 3 * 3 > 5 jest prawdziwe
```

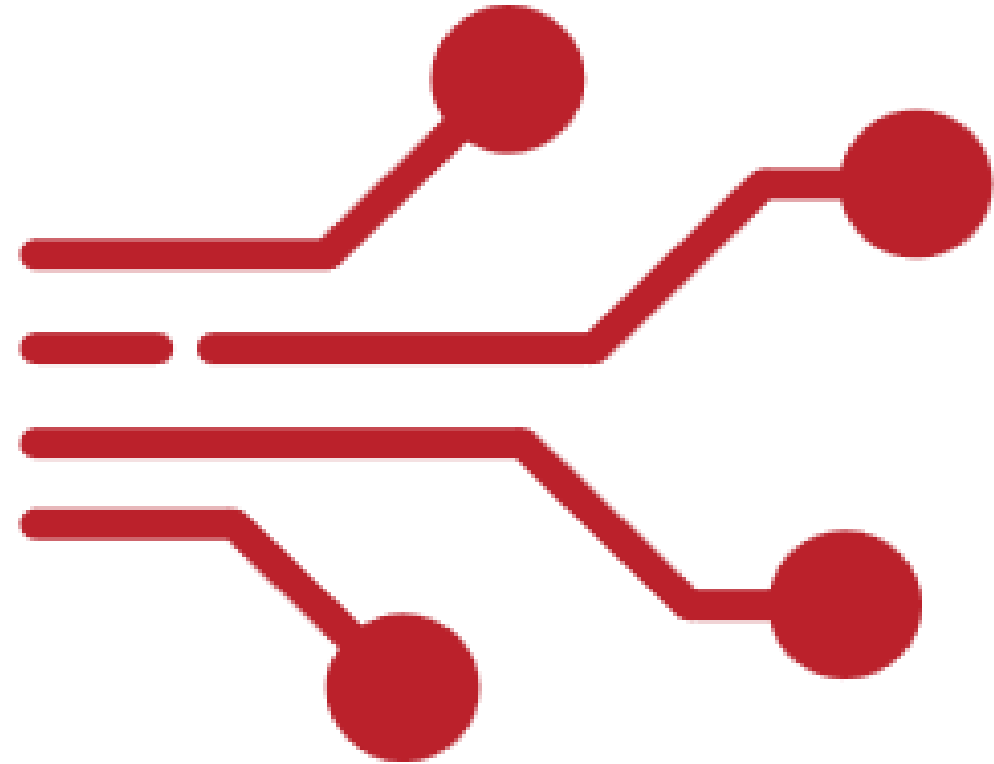
Skróty klawiszowe

- **ctrl-c** - przerwanie polecenia.
- **ctrl-d** - koniec wprowadzania danych.
- **ctrl-s** - zatrzymaj wyświetlanie na ekran.
- **ctrl-q** - wznów wyświetlanie na ekran.
- **ctrl-z** - zawieś wykonywanie polecenia.

Polecenie stty -a

```
toor@Mikasa:~$ stty -a
speed 38400 baud; rows 28; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany -imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echopr
echoctl echoke -flusho -extproc
```

Programowanie w bash



47

Czego się spodziewać?

- Programy w bash umożliwiają wprowadzenie modyfikacji środowiska, które wcześniej były niemożliwe do osiągnięcia samą konfiguracją.
- Część technik programowania w bash stanowi rozwinięcie technik dostosowywania środowiska powłoki.
- Inne elementy przypominają składowe tradycyjnych języków programowania.

Przygotowanie do uruchomienia

- Ustawienie odpowiedniego shebang: **#!/bin/bash**.
- Nadanie atrybutu wykonywalności: **chmod +x script.sh**.
- Ewentualne nadanie dodatkowych atrybutów lub praw własności.
- **. script.sh** - wykonuje skrypt w bieżącej powłoce.
- **./script.sh** - tworzy podpowłokę i oczekuje na zakończenie wykonania skryptu.
- **./script.sh &** - tworzy podpowłokę i nie oczekuje na zakończenie wykonania skryptu.

49

Funkcje 1

- Funkcja to „skrypt w skrypcie” czyli fragment kodu powłoki zdefiniowany pod nazwą i przechowywany w pamięci, który można później wywołać.
- Funkcje znacząco zwiększają możliwości programistyczne powłoki.
- Funkcje działają szybciej niż zewnętrzne skrypty, ponieważ ich kod jest już w pamięci powłoki w momencie wywołania.
- Funkcje są idealne do dzielenia długich skryptów na modułowe fragmenty kodu, które łatwiej rozwijać i utrzymywać.

51

Funkcje 2

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

```
user_name ()  
{  
    who | cut -f1 -d\  
}
```

```
user_name
```

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

```
function user_name  
{  
    who | cut -f1 -d\  
}
```

```
user_name
```

Funkcje 3

- Istnieją dwie ważne różnice między funkcjami a skryptami:
 - Funkcje nie działają w osobnych procesach, w przeciwieństwie do skryptów wywoływanych po nazwie.
 - Uruchamianie funkcji jest semantycznie podobne do wykonywania skryptu wywołanego poleceniem `source`.
- Jeśli funkcja ma tę samą nazwę co skrypt lub program wykonywalny, funkcja ma pierwszeństwo.
- Możliwe jest usunięcie z pamięci deklaracji funkcji z wykorzystaniem polecenia: **`unset -f nazwa_funkcji`**.

53

Zmienne w programowaniu powłoki 1

- Powłoka ustawia część zmiennych środowiskowych automatycznie podczas logowania.
- Istnieją wbudowane zmienne, które są kluczowe w programowaniu powłoki.
- Zmienne są definiowane poprzez nazwę i inicjalizowane poprzez przypisanie wartości.

Zmienne w programowaniu powłoki 2

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

```
hello="Hello Bash!"
```

```
echo $hello
```

```
toor@Mikasa:~$ ./script.sh  
Hello Bash!
```

56

Zmienne w programowaniu powłoki 3

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

Uwaga na spacje!

```
hello = "Hello Bash!"
```

```
echo $hello
```

```
toor@Mikasa:~$ ./script.sh
./script.sh: linia 3: Hello Bash!: nie znaleziono polecenia
```

57

Zmienne (parametry) pozycyjne 1

- Zmienne pozycyjne mają nazwy 1, 2, 3 itd., a odwołujemy się do nich jako **\$1**, **\$2**, **\$3**.
- Zmienna **\$0** zawiera nazwę skryptu (polecenie użyte do jego uruchomienia).
- Dwie specjalne zmienne, ***** i **@**, zawierają wszystkie parametry pozycyjne (poza 0).
- **\${*}** to jeden ciąg znaków z wszystkimi parametrami.
- **\${@}** jest równoważne **"\${1}"** **"\${2}"** ... **"\${N}"** – to N osobnych, podwójnie cytowanych ciągów, oddzielonych spacjami.
- Jeśli nie ma parametrów pozycyjnych, **\${@}** rozwija się do pustego ciągu.

58

Zmienne (parametry) pozycyjne 2

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

```
echo $0
```

```
echo $1
```

```
echo $2
```

```
echo ${*}
```

```
echo ${@}
```

```
toor@Mikasa:~$ ./script.sh Hello Bash!  
./script.sh  
Hello  
Bash!  
Hello Bash!  
Hello Bash!
```

59

Zmienne (parametry) pozycyjne 3

- Zmienna # przechowuje liczbę parametrów pozycyjnych (jako ciąg znaków).
- Wszystkie zmienne pozycyjne są tylko do odczytu i nie można nadawać im nowych wartości w skryptach.
- Zmienne pozycyjne, do których się odwołamy a nie zostały przekazane przez nie wartości zawierają pusty łańcuch.

Zmienne (parametry) pozycyjne w funkcji 1

- **\$*** i **\$@** to dwa specyficzne i często mylące mechanizmy powłoki związane z parametrami pozycyjnymi.
- Elementy **"\$*"** są rozdzielane pierwszym znakiem zmiennej **IFS**, a nie tylko spacjami, co daje większą elastyczność formatu wyjścia.
- Przykład: ustawienie **IFS=","** i polecenie **echo "\$*"** pozwala wypisać parametry oddzielone przecinkami.
- Zmiana **IFS** w skrypcie jest ryzykowna, ale może być akceptowalna, jeśli reszta skryptu od niej nie zależy.
- **"\$@"** zachowuje się jak N osobnych, podwójnie cytowanych ciągów pozwala ponownie użyć argumentów jako osobnych wartości.

61

Zmienne (parametry) pozycyjne w funkcji 2

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

```
IFS=,
```

```
echo "$@"
```

```
toor@Mikasa:~$ ./script.sh Hello Bash!  
Hello, Bash!
```

62

Zmienne (parametry) pozycyjne w funkcji 3

- Funkcje powłoki używają parametrów pozycyjnych i specjalnych zmiennych takich jak * i # tak samo jak skrypty.
- W jednym skrypcie zwykle definiuje się wiele funkcji, z których każda powinna obsługiwać własne argumenty.
- Każda funkcja ma własne kopie parametrów pozycyjnych i zmiennych specjalnych i są one lokalne dla funkcji.
- Pozostałe zmienne tworzone w funkcjach nie są lokalne domyślnie, a są globalne i ich wartości są widoczne w całym skrypcie.

Zmienne (parametry) pozycyjne w funkcji 4

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

```
function testuj_parametry  
{  
    echo "W funkcji \$1 = $1, \$2 = $2"  
}
```

```
echo "W skrypcie \$1 = $1, \$2 = $2"
```

```
testuj_parametry Hello
```

```
toor@Mikasa:~$ ./script.sh Witaj w skrypcie!  
W skrypcie $1 = Witaj, $2 = w  
W funkcji $1 = Hello, $2 =
```

64

Zmienne globalne

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

```
globalna="G"
```

```
function testuj_zasieg
```

```
{
```

```
    lokalna="L"
```

```
    echo "W funkcji:"
```

```
    echo "\$globalna = $globalna"
```

```
    echo "\$lokalna = $lokalna"
```

```
}
```

```
echo "W skrypcie:"
```

```
echo "\$globalna = $globalna"
```

```
echo "\$lokalna = $lokalna"
```

```
testuj_zasieg
```

```
echo "W skrypcie:"
```

```
echo "\$globalna = $globalna"
```

```
echo "\$lokalna = $lokalna"
```

```
toor@Mikasa:~$ ./script.sh
W skrypcie:
$globalna = G
$lokalna =
W funkcji:
$globalna = G
$lokalna = L
W skrypcie:
$globalna = G
$lokalna = L
```

65

Zmienne lokalne 1

- Modyfikator **local** wewnątrz definicji funkcji powoduje, że wskazane zmienne stają się lokalne dla tej funkcji.
- Zmienne lokalne są kluczowe przy pisaniu większych programów, bo zapewniają niezależność funkcji od programu głównego i innych funkcji.

Zmienne lokalne 2

```
#!/bin/bash
globalna="G"

function testuj_zasieg
{
    local lokalna="L"
    echo "W funkcji:"
    echo "\$globalna = $globalna"
    echo "\$lokalna = $lokalna"
}

echo "W skrypcie:"
echo "\$globalna = $globalna"
echo "\$lokalna = $lokalna"
testuj_zasieg
echo "W skrypcie:"
echo "\$globalna = $globalna"
echo "\$lokalna = $lokalna"
```

Projekt współfinansowany ze środków Unii Europejskiej w ramach:
FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS)
"POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

```
toor@Mikasa:~$ ./script.sh
W skrypcie:
$globalna = G
$lokalna =
W funkcji:
$globalna = G
$lokalna = L
W skrypcie:
$globalna = G
$lokalna =
```

67



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Drobne uwagi

- **`$varname`** to składnia uproszczona, pełna postać to **`${varname}`**.
- Pełna składnia jest konieczna m.in. przy odwołaniu do parametrów pozycyjnych powyżej 9, np. **`${10}`** zamiast **`$10`**.
- Bez nawiasów klamrowych powłoka może „skleić” nazwę zmiennej z następnymi znakami, np. echo **`$UID_`** szuka zmiennej **`UID_`** i zwykle nic nie wypisze.
- Aby uzyskać wartość zmiennej UID plus podkreślenie, trzeba użyć echo **`${UID}_`**.
- Nawiasy klamrowe można pominąć tylko wtedy, gdy po nazwie zmiennej występuje znak inny niż litera, cyfra lub podkreślenie.

68

Zmienne lokalne 3

- Modyfikator **local** wewnątrz definicji funkcji powoduje, że wskazane zmienne stają się lokalne dla tej funkcji.
- Zmienne lokalne są kluczowe przy pisaniu większych programów, bo zapewniają niezależność funkcji od programu głównego i innych funkcji.

Operatory łańcuchowe

- Składnia z nawiasami klamrowymi $\{...\}$ umożliwia używanie operatorów tekstowych na zmiennych.
- Operatory tekstowe pozwalają modyfikować wartości zmiennych bez pisania rozbudowanych programów ani używania zewnętrznych narzędzi systemu.
- Można ich używać, nawet jeśli nie zna się jeszcze bardziej zaawansowanych możliwości programowania w powłoce.

Operatory tekstowe 1

- Dzięki operatorom tekstowym można m. in.:
 - upewniać się, że zmienne istnieją (są zdefiniowane i niepuste),
 - ustawiać wartości domyślne,
 - wychwytywać błędy wynikające z braku ustawionych zmiennych,
 - usuwać fragmenty wartości zmiennych pasujące do wzorca.

Operatory tekstowe 2

- Składnia operatorów tekstowych polega na wstawieniu specjalnych znaków (operatora) między nazwą zmiennej a prawy nawias klamrowy.
- Ewentualny argument operatora umieszczany jest na prawo od tego operatora, wewnątrz $\${...}$.

Operatory tekstowe 3

- **`${var:-word}`** - jeżeli zmienna **var** istnieje i nie jest null zwraca tą wartość, w przeciwnym razie zwraca **word**.
- **`${var:=word}`** - jeżeli zmienna **var** istnieje i nie jest null zwraca tą wartość, w przeciwnym razie przypisuje do niej **word** i wtedy zwraca.
- **`${var:?msg}`** - jeżeli zmienna **var** istnieje i nie jest null zwraca tą wartość, w przeciwnym wyświetla **var: msg** i przerywa wykonanie.

Operatory tekstowe 4

- Kolejny typ operatorów tekstowych służy do dopasowywania fragmentów wartości zmiennej do wzorców.
- Wzorce to ciągi znaków, które mogą zawierać wildcardy: *, ? oraz [] (zbiory i zakresy znaków).

Operatory tekstowe 5

- **`${var#pattern}`** - usuwa najkrótszą część z początku wartości pasującą do wzorca, zwraca resztę.
- **`${var##pattern}`** - usuwa najdłuższą część z początku wartości pasującą do wzorca, zwraca resztę.
- **`${var%pattern}`** - usuwa najkrótszą część z końca wartości pasującą do wzorca, zwraca resztę.
- **`${var%%pattern}`** - usuwa najdłuższą część z końca wartości pasującą do wzorca, zwraca resztę.
- **`${var/pattern/string}`** - zastępuje pierwsze dopasowanie wzorca w wartości zmiennej podanym ciągiem **string**.
- **`${var//pattern/string}`** - zastępuje wszystkie dopasowania wzorca w wartości zmiennej ciągiem **string**.

75

Operatory tekstowe 6

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

```
path=${PWD}"/"${0}
```

```
echo ${path}
```

```
echo ${path#*/}
```

```
echo ${path##*/}
```

```
echo ${path%.*}
```

```
echo ${path%.*}
```

```
echo ${path/toor/root}
```

```
toor@Mikasa:~/scripts$ ./patterns.example.sh
/home/toor/scripts/./patterns.example.sh
toor/scripts/./patterns.example.sh
patterns.example.sh
/home/toor/scripts/./patterns.example
/home/toor/scripts/
/home/root/scripts/./patterns.example.sh
```

76

Operatory tekstowe 7

- Bash udostępnia rozszerzony zestaw operatorów dopasowania wzorców, jeśli włączona jest opcja **shopt -s extglob**.
- Każdy z tych operatorów przyjmuje jeden lub więcej wzorców, zazwyczaj ciągi znaków, rozdzielonych pionową kreską | (tzw. pattern-list).

Operatory tekstowe 8

- ***(patternlist)** – dopasowuje zero lub więcej wystąpień podanych wzorców.
- **+(patternlist)** – dopasowuje jedno lub więcej wystąpień podanych wzorców.
- **?(patternlist)** – dopasowuje zero lub jedno wystąpienie podanych wzorców.
- **@(patternlist)** – dopasowuje dokładnie jedno z podanych wzorców.
- **!(patternlist)** – dopasowuje wszystko z wyjątkiem elementów pasujących do któregoś z podanych wzorców.

78

Podstawienie poleceń 1

- W powłoce można wprowadzać wartości do zmiennych na trzy sposoby: przez przypisanie, przez argumenty pozycyjne oraz przez podstawianie poleceń.
- Podstawianie poleceń pozwala traktować standardowe wyjście polecenia tak, jakby było wartością zmiennej.
- Składnia podstawiania poleceń to: **\$(polecenie)**.

Podstawienie poleceń 2

- Polecenie wewnątrz nawiasów jest wykonywane, a wszystko, co wypisze na standardowe wyjście, staje się wynikiem wyrażenia.
- Konstrukcje te można zagnieżdżać, tzn. polecenie wewnątrz \$(...)\$ może samo zawierać kolejne podstawienia poleceń.
- **Podstawienie poleceń usuwa znaki końca linii!**

Podstawienie poleceń 3

`#!/bin/bash`

`ls=$(cal)`

`echo $ls`

```
toor@Mikasa:~/scripts$ cal
      Luty 2026
ni po wt śr cz pi so
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28

toor@Mikasa:~/scripts$ ./example.sh
Luty 2026 ni po wt śr cz pi so 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 2
0 21 22 23 24 25 26 27 28
```

81

Operacje na stosie katalogów 1

- Polecenia **pushd** i **popd** służą do wygodnej nawigacji między katalogami z użyciem stosu katalogów.
- Polecenie **pushd** zmienia bieżący katalog na wskazany i jednocześnie odkłada poprzedni katalog na stos.
- Polecenie **popd** usuwa katalog ze szczytu stosu i ustawia go jako bieżący katalog roboczy (wracasz „tam, skąd przyszedłeś”).
- Typowy wzorzec użycia: najpierw **pushd /jakis/katalog**, wykonanie poleceń, a potem **popd**, aby wrócić do wcześniejszego katalogu.
- Polecenia te są szczególnie przydatne w skryptach, gdy trzeba przechodzić przez wiele katalogów i na końcu wrócić do punktu startowego bez ręcznego śledzenia ścieżek.

82

Operacje na stosie katalogów 2

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

```
echo $PWD
```

```
mkdir src
```

```
pushd src
```

```
echo $PWD
```

```
touch script.sh
```

```
popd
```

```
echo $PWD
```

```
echo $(ls src)
```

```
toor@Mikasa:~/scripts$ ./example.sh
/home/toor/scripts
~/scripts/src ~/scripts
/home/toor/scripts/src
~/scripts
/home/toor/scripts
script.sh
```

83

Opcje wywołania skryptu

- Parametry pozycyjne (\$1, \$2, ..., \$*, \$#) zawierają całą informację z linii poleceń przekazanej do skryptu lub funkcji.
- Standardowa forma poleceń w systemie zawiera opcje, których może być wiele i są identyfikowane przez jej nazwę z myślnikiem.
- Napisanie skryptu zachowujący się jak typowe polecenia powłoki, trzeba jawnie obsłużyć nazwane opcje w kodzie.
- Można w tym celu wykorzystać pętlę po parametrach z odpowiednią kombinacją instrukcji warunkowych.

Operacja shift

- W bashu wbudowane polecenie **shift** służy do przesuwania parametrów pozycyjnych (**\$1**, **\$2**, ..., **\$#**) w lewo.
- Po wywołaniu **shift** bez argumentu dotychczasowe **\$2** staje się **\$1**, **\$3** staje się **\$2**, itd., a stare **\$1** jest tracone.
- Możesz podać liczbę to **shift N** to wtedy przesuwasz parametry o N pozycji w lewo; parametry od **\$1** do **\$N** zostają odrzucone, **\$N+1** staje się **\$1** itd.
- Jeśli N jest pominięte, domyślnie przyjmowana jest wartość 1.
- N nie może być ujemne, a jeśli jest większe niż liczba dostępnych parametrów (**\$#**), **shift** zwróci błąd (kod wyjścia $\neq 0$).

85

Argumenty nazwane 1

- Powłoka dostarcza wbudowany mechanizm obsługi wielu złożonych opcji bez opisanych wcześniej ograniczeń i jest nim polecenie **getopts**.
- Zwykle wykorzystuje się jako warunek w pętli **while** obsługującej opcje z linii poleceń.
- Podajesz specyfikację, które opcje są dozwolone i które z nich wymagają własnych argumentów.
- Na tej podstawie **getopts** przygotowuje ciało pętli do przetworzenia każdej opcji po kolei.

86

Argumenty nazwane 2

- Polecenie **getopts** przyjmuje dwa argumenty:
 - pierwszy to łańcuch zawierający litery i dwukropki, każda litera oznacza prawidłową opcję, a litera z dwukropkiem po niej oznacza, że ta opcja wymaga argumentu czyli "ab:c": -a bez argumentu, -b z argumentem, -c bez argumentu,
 - drugi to nazwa zmiennej, do której getopts zapisze kolejną znaną opcję bez poprzedzającego myślnika.

Argumenty nazwane 3

- Polecenie **getopts** kolejno pobiera opcje z linii poleceń i w każdej iteracji przypisuje nazwę opcji do zmiennej podanej jako drugi argument, a wartość argumentu opcji (jeśli wymagana) trafia do zmiennej wbudowanej **OPTARG**.
- Dopóki są jeszcze opcje do przetworzenia, **getopts** zwraca kod wyjścia 0, a gdy opcje się skończą, zwraca 1, co powoduje wyjście z pętli while.

Argumenty nazwane 4

- Jeśli nie zaczniesz łańcucha opcji od dwukropka, **getopts** przy nieprawidłowej opcji wypisze swój własny komunikat błędu.
- Jeśli zaczniesz łańcuch opcji od dwukropka (np. ":ab:c"), **getopts** przestaje wypisywać wbudowane komunikaty błędów jednocześnie ustawia zmienną opcji na **?**, więc możesz to wykryć w instrukcji **case**.
- W ten sposób możesz wypisać własny komunikat błędu, korzystając z **OPTARG**, żeby wiedzieć, która opcja była nieprawidłowa.

89

Argumenty nazwane 5

```
while getopts ":ab:c" o; do
    case "$o" in
        a)  echo "opcja -a" ;;
        b)  echo "opcja -b argument $OPTARG" ;;
        c)  echo "opcja -c" ;;
        \?) echo "Błędna opcja: -$OPTARG" ;;
        :)  echo "Brak argumentu -$OPTARG" ;;
    esac
done
```

90

Argumenty nazwane 6

```
toor@Mikasa:~/scripts$ ./example.sh
toor@Mikasa:~/scripts$ ./example.sh -a
opcja -a
toor@Mikasa:~/scripts$ ./example.sh -b
Brak argumentu -b
toor@Mikasa:~/scripts$ ./example.sh -b Bash
opcja -b argument Bash
toor@Mikasa:~/scripts$ ./example.sh -c
opcja -c
toor@Mikasa:~/scripts$ ./example.sh -d
Błędna opcja: -d
```

91

Kontrola przepływu w skrypcie

- Konstrukcje sterowania przepływem pozwalają decydować, które fragmenty programu się wykonają i ile razy.
- Działania te są zależne od spełnienia określonych warunków np.: wartości zmiennych, wyników poleceń itp.
- Większość dotychczasowych przykładów skryptów była prostą listą poleceń, bez rozgałęzień czy pętli.
- Powłoka bash w swoich skryptach udostępnia pełen zestaw konstrukcji sterujących.

92

If/else 1

- Najprostszą konstrukcją sterowania przepływem w bash jest instrukcja **if**.
- Używamy warunków, gdy chcemy zdecydować, czy coś wykonać na podstawie wyniku testu (prawda/fałsz).
- Warunki mogą sprawdzać m.in. wartości zmiennych powłoki, cechy plików oraz to, czy polecenia zakończyły się powodzeniem.
- Powłoka udostępnia rozbudowany zestaw wbudowanych testów dopasowanych do typowych zadań w skryptach.

93

If/else 2

```
if warunek  
then  
operacje...  
[elif warunek  
then  
operacje...]  
[else  
operacje...]  
fi
```

94

If/else 3

- Najprostsza postać **if** (bez **elif** i **else**) wykonuje blok poleceń tylko wtedy, gdy warunek jest prawdziwy.
- Dodanie części **else** pozwala wykonać jeden zestaw poleceń, gdy warunek jest prawdziwy, lub inny, gdy jest fałszywy.
- Klauzule **elif** („else if”) umożliwiają sprawdzanie wielu warunków po kolei i wybór między wieloma blokami poleceń.
- Można użyć dowolnej liczby **elif**, a końcowa część **else** pełni funkcję działania „ostatniej szansy”.

95

If/else 4

- W bash jedyna istotna różnica względem języków takich jak C czy Pascal polega na tym, że „warunek” w **if** jest listą poleceń, a nie typowym wyrażeniem logicznym.
- Prawdziwość warunku jest określana na podstawie kodu zakończenia (exit status) polecenia.
- Każde polecenie powłoki zwraca do powłoki kod całkowity: 0 zwykle oznacza powodzenie, a wartości 1–255 błąd.

If/else 5

- Instrukcja **if** sprawdza exit status ostatniego polecenia w liście po słowie kluczowym.
- Jeśli status wynosi 0, warunek jest traktowany jako prawdziwy.
- Jeśli jest różny od 0, warunek jest fałszywy.
- Ta sama zasada dotyczy warunków po słowach **elif**.

If/else 6

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

```
if cd $1 2> /dev/null  
then  
    echo "Przeszliśmy do katalogu ${1}"  
else  
    echo "Katalog ${1} nie istnieje"  
fi
```

```
echo "Bieżący katalog: $PWD"
```

Projekt współfinansowany ze środków Unii Europejskiej w ramach:
FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS)
"POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

```
toor@Mikasa:~/scripts$ ./example.sh src  
Przeszliśmy do katalogu src  
Bieżący katalog: /home/toor/scripts/src  
toor@Mikasa:~/scripts$ ./example.sh abc  
Katalog abc nie istnieje  
Bieżący katalog: /home/toor/scripts
```

return

- Instrukcja **return** powoduje zakończenie bieżącej funkcji z kodem wyjścia równym przekazanej w niej wartości.
- Argument jest opcjonalny i jeśli go nie podamy, użyty zostanie kod wyjścia ostatniego wykonanego polecenia w funkcji.
- Funkcja, która kończy się bez **return**, zwraca kod wyjścia ostatniego polecenia w swoim ciele.
- Instrukcję **return** można stosować wyłącznie wewnątrz funkcji oraz w skryptach uruchomionych przez **. nazwa**.

exit

- Instrukcja **exit** kończy działanie całego skryptu (całej powłoki skryptowej), niezależnie od głębokości zagnieżdżenia w funkcjach.
- Analogicznie do return zwraca przekazany argument lub kod wyjścia ostatniego polecenia.

100

Łączenie kodów wyjścia

- W bash **&&** i **||** służą do łączenia poleceń na podstawie ich kodu wyjścia i tego czy wykonanie polecenia „się udało”.
- **&&** (logiczne „i”) oznacza, że drugie polecenie wykona się tylko wtedy, gdy pierwsze zakończy się sukcesem (kod wyjścia 0), a sekcja **then** zostanie wykonana tylko, jeżeli drugie polecenie zakończy się sukcesem.
- **||** (logiczne „lub”): drugie polecenie wykona się tylko wtedy, gdy pierwsze zakończy się błędem (kod wyjścia $\neq 0$), co oznacza, że sekcja **then** wykona się jeżeli którekolwiek poleceń zakończy się sukcesem.

101

Testy warunków 1

- Instrukcje **if** w bash mogą testować tylko kody wyjścia (exit status), ale dzięki specjalnym konstrukcjom, które zwracają kod wyjścia da się sprawdzać również inne warunki.
- Służą do tego dwie formy składni: `[ ... ]` oraz nowsza `[[ ... ]]`.
- Wersja `[[ ... ]]` działa podobnie jak `[ ... ]`, ale wewnątrz nawiasów nie ma podziału na słowa ani rozwijania wzorców ścieżek (globbingu), co czyni ją bezpieczniejszą w wielu przypadkach.

102

Testy warunków 2

- Za pomocą tych konstrukcji można m.in. sprawdzać cechy plików (czy istnieją, typ, uprawnienia, właściciel), porównywać, który plik jest nowszy oraz wykonywać porównania tekstowe.
- `[ i ]` tworzą zwykłe polecenie, które jedynie zwraca kod wyjścia mówiący, czy warunek jest prawdziwy; wymagane są spacje po `[ i` przed `]`, np. `[ warunek ]`.

Testy warunków - porównanie łańcuchów

- **str1 = str2** - takie same
- **str1 != str2** - różne
- **str1 < str2** - str1 jest mniejszy niż str2
- **str1 > str2** - str1 jest większy niż str2
- **-n str** - łańcuch nie jest pusty (niezerowa długość)
- **-z str** - łańcuch jest pusty (zerowa długość)

Testy warunków - sprawdzanie plików 1

- **-a element**, **-e element** - istnienie pliku, katalogu lub innego elementu w drzewie plików
- **-d katalog** - istnienie katalogu
- **-f plik** - istnienie (zwykłego) pliku

Testy warunków - sprawdzanie plików 2

- **-r element** - element z prawem odczytu
- **-w element** - element z prawem zapisu
- **-x element** - element z prawem wykonania/przeszukania
- **-s element** - element jest niepusty

Testy warunków - sprawdzanie plików 3

- **-N element** - element został zmodyfikowany od czasu ostatniego odczytu
- **-O element** - sprawdzenie czy jesteś właścicielem elementu
- **-G element** - sprawdzenie czy grupa, do której należysz jest właścicielem elementu
- **element1 -nt element2** - element1 został zmodyfikowany później niż element2
- **element1 -ot element2** - element1 został zmodyfikowany wcześniej niż element2

107

Testy warunków - sprawdzanie plików 4

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

```
if [ -d $1 ]
```

```
then
```

```
    echo "${1} to katalog"
```

```
elif [ -f $1 ]
```

```
then
```

```
    echo "${1} to plik"
```

```
else
```

```
    echo "${1} nie jest ani plikiem ani katalogiem"
```

```
fi
```

```
toor@Mikasa:~/scripts$ ./example.sh example.sh
example.sh to plik
toor@Mikasa:~/scripts$ ./example.sh src
src to katalog
toor@Mikasa:~/scripts$ ./example.sh abc
abc nie jest ani plikiem ani katalogiem
```

108

Łączenie warunków 1

- W wyrażeniach warunkowych w nawiasach kwadratowych [] możesz łączyć warunki operatorami logicznymi **&&** oraz **|**, tak jak w zwykłych poleceniach powłoki.
- Przykład:
if [condition] && [condition]; then
- Możliwe jest łączenie poleceń powłoki z wyrażeniami warunkowymi za pomocą operatorów logicznych.
- Przykład:
 - **if command && [condition]; then**

109

Łączenie warunków 2

- Możesz zanegować wartość logiczną wyrażenia warunkowego poprzedzając je znakiem **!**.
- Da się budować złożone wyrażenia logiczne, grupując warunki nawiasami **()**.
- Nawiasy muszą być poprzedzone ukośnikami odwrotnymi **\(\)**, żeby powłoka nie potraktowała ich specjalnie.
- W testach możesz używać dodatkowych operatorów logicznych **-a** (i) oraz **-o** (lub).
- Operatory **-a** i **-o** działają podobnie do **&&** i **||** używanych z kodami wyjścia poleceń, ale są dostępne tylko wewnątrz wyrażenia testowego.

110

Testowanie liczb

- W powłoce są dostępne również testy arytmetyczne, różne od porównań napisów.
- Operatory $<$ i $>$ przy porównywaniu ciągów znaków działają leksykograficznie (słownikowo), a nie numerycznie.
- Na przykład napis "6" jest leksykograficznie większy niż "57", tak jak "p" jest większe niż "ox".
- Jednak w sensie liczb całkowitych $6 < 57$, więc wynik porównania zależy od tego, czy traktujemy wartości jako napisy, czy liczby.

111

Testowanie liczb całkowitych

- **-lt** - mniejsze niż
- **-le** - mniejsze lub równe
- **-eq** - równe
- **-ge** - większe lub równe
- **-gt** - większe niż
- **-ne** - różne

112

Case 1

- W językach takich jak C/Java konstrukcja case/switch służy do testowania prostych wartości (np. liczb całkowitych, znaków).
- W bashu konstrukcja **case** pozwala testować napisy względem wzorców.
- Podobnie jak w klasycznym **if-then-else**, **case** umożliwia opisanie wielu gałęzi warunkowych, ale w znacznie bardziej zwężły i czytelny sposób.

113

Case 2

```
case wyrażenie in  
    wzorzec1 )  
        polecenia ;;  
    wzorzec2 )  
        polecenia ;;  
esac
```

114

Case 3

- W językach takich jak C/Java konstrukcja case/switch służy do testowania prostych wartości (np. liczb całkowitych, znaków).
- W bashu konstrukcja **case** pozwala testować napisy względem wzorców.
- Podobnie jak w klasycznym **if-then-else**, **case** umożliwia opisanie wielu gałęzi warunkowych, ale w znacznie bardziej zwężły i czytelny sposób.

115

Case 4

```
#!/bin/bash

for f in *
do
    case $f in
        *.mp4 )
            mpeg2avi $f ;;
        *.mov )
            mov2avi $f ;;
        * )
            echo "Format niewspierany"
    ;;
    esac
done
```

116

Select 1

- Konstrukcja **select** występuje tylko w Korn shellu i bashu, i nie ma odpowiednika w klasycznych językach programowania.
- Służy do łatwego tworzenia prostych menu tekstowych, przy zwartej składni wykonuje sporo pracy „pod spodem”.
- Składnia jest taka sama jak dla **for**, z tą różnicą, że używasz słowa kluczowego **select**.

117

Select 2

- Działanie select:
 - Generuje menu z elementów list, ponumerowanych kolejnymi liczbami.
 - Pyta użytkownika o numer wyboru (prompt).
 - Zapisuje wybraną pozycję do zmiennej name, a numer wyboru do zmiennej wbudowanej **REPLY**.
 - Wykonuje instrukcje w bloku **do ... done**.
 - Powtarza cały proces w pętli w nieskończoność (dopóki nie przerwiesz np. break lub Ctrl+C).

118

Select 3

- Wbudowana zmienna powłoki PS3 zawiera tekst zachęty (prompt), którego używa konstrukcja `select`.
- Domyślna wartość PS3 to mało użyteczne "#?".
- Dlatego pierwsza linia w kodzie zawierającym **`select`** zwykle ustawia PS3 na bardziej sensowny komunikat.

119

Select 4

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

```
PS3="wybierz katalog: "  
select d in $1/*  
do  
    ls $d  
done
```

```
toor@Mikasa:~/scripts$ ./example.sh /var  
1) /var/backups      7) /var/log          13) /var/snap  
2) /var/cache        8) /var/mail         14) /var/spool  
3) /var/crash        9) /var/metrics     15) /var/tmp  
4) /var/lib          10) /var/nfs         16) /var/www  
5) /var/local        11) /var/opt  
6) /var/lock         12) /var/run  
wybierz katalog: 2  
adduser      cups      fwupd      private  
apparmor     cups-browsed  fwupdmgr   snapd  
app-info     debconf      ldconfig   swcatalog  
apt          dictionaries-common  man  
cracklib     fontconfig   PackageKit  
wybierz katalog:
```

120

Pętla for 1

- Możliwość wykonania testów dla wielu plików, a nie tylko jednego wymaga zrealizowania iteracji.
- Do takiego powtarzania działań w bashu używa się konstrukcji pętli, w szczególności pętli **for**.
- Pętla for pozwala powtórzyć fragment kodu określoną liczbę razy.
- Przy każdej iteracji pętli ustawiana jest zmienna pętli na kolejny element, dzięki czemu każda iteracja może wykonywać następną operację.

121

Pętla for 2

- Standardowa pętla **for** w shellu nie pozwala bezpośrednio podać liczby iteracji ani zakresu wartości.
- Zamiast tego przyjmuje stałą listę wartości, po których iteruje, np. listę plików, słów lub wynik rozwinięcia wzorca.
- Od Basha 2.0 jest też dostępna arytmetyczna odmiana pętli pozwalająca na użycie zmiennych licznikowych.

122

Pętla for 3

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

```
for f in $1/*  
do  
    echo $f  
done
```

```
toor@Mikasa:~/scripts$ ./example.sh /var  
/var/backups  
/var/cache  
/var/crash  
/var/lib  
/var/local  
/var/lock  
/var/log  
/var/mail  
/var/metrics  
/var/nfs  
/var/opt  
/var/run  
/var/snap  
/var/spool  
/var/tmp  
/var/www
```

123

Pętle while i until 1

- Obie pozwalają wielokrotnie wykonywać fragment kodu dopóki (**while**) lub aż (**until**) określony warunek stanie się prawdziwy.
- Są podobne do odpowiednich konstrukcji z C (while, do/while).
- Pętle **while** i **until** są szczególnie przydatne w połączeniu z arytmetyką całkowitą, wejściem/wyjściem do zmiennych oraz przetwarzaniem argumentów z linii poleceń.

124

Pętle while i until 2

- Jedyna różnica między **while** a **until** dotyczy sposobu interpretacji warunku.
- W pętli **while** kod wykonuje się tak długo, jak warunek jest prawdziwy (polecenie zwraca kod wyjścia 0).
- W pętli **until** kod wykonuje się tak długo, jak warunek jest fałszywy (polecenie zwraca kod wyjścia $\neq 0$).

125

Pętle while i until - składnia

```
while warunek  
do  
    polecenia  
done
```

```
until warunek  
do  
    polecenia  
done
```

126

Declare 1

- Słowo kluczowe **declare** w Bashu to wbudowane polecenie do definiowania zmiennych i funkcji oraz nadawania im atrybutów.
- Służy do deklarowania zmiennych i jednoczesnego przypisywania im wartości (**declare NAZWA=wartość**).
- Pozwala ustawiać typ/atribut zmiennej.

127

Declare 2

- Użyte w funkcji sprawia, że zmienna jest lokalna dla tej funkcji (podobnie jak **local**).
- Z opcją **-g** wymusza utworzenie zmiennej globalnej, nawet gdy zostało użyte wewnątrz funkcji.
- Atrybuty można też zdejmować, używając **+** zamiast **-** (np. **declare +r VAR**).

128

Declare 3

- Typy/atributy zmiennych:
 - **-a** – zwykła tablica indeksowana.
 - **-A** – tablica asocjacyjna (klucze tekstowe).
 - **-i** – zmienna całkowitoliczbowa, wyrażenia są obliczane arytmetycznie.
 - **-r** – zmienna tylko do odczytu (readonly, nie da się jej później zmienić).

Declare 4

- Typy/atributy zmiennych
 - **-x** – zmienna eksportowana do środowiska (dostępna w procesach potomnych).
 - **-n** – referencja do innej zmiennej (nameref).
 - **-g** – wymusza zasięg globalny (nawet jeśli użyte wewnątrz funkcji).
 - **-t** – ustawienie atrybutu „trace” (rzadko używane; do debugowania).

130

Declare 5

- Atrybuty dotyczące formatowania tekstu
 - **-l** – automatyczna zamiana wartości na małe litery.
 - **-u** – automatyczna zamiana wartości na wielkie litery.
- Atrybuty związane z funkcjami / wyświetlaniem
 - **-f** – działaj na funkcjach (ogranicz do nazw i definicji funkcji).
 - **-F** – pokazuj tylko nazwy funkcji (plus numery linii, jeśli dostępne).
 - **-p** – wypisz atrybuty i wartości wskazanych zmiennych (lub wszystkich, jeśli nie podasz nazw).

131

Declare - przykład 1

```
#!/usr/bin/env bash
```

```
# mienna całkowitoliczbowa (-i)
```

```
declare -i counter=10
```

```
counter=counter+5          # zostanie policzone arytmetycznie
```

```
echo "counter = $counter"  # 15
```

```
# Zmienna tylko do odczytu (-r)
```

```
declare -r VERSION="1.0.0"
```

```
echo "VERSION = $VERSION"
```

```
VERSION="2.0.0"           # błąd: tylko do odczytu
```

132

Declare - przykład 2

```
#!/usr/bin/env bash
```

```
# Tablica indeksowana (-a)
```

```
declare -a fruits=("apple" "banana" "cherry")
```

```
fruits[3]="orange"
```

```
echo "fruits = ${fruits[@]}"
```

```
# Tablica asocjacyjna (-A)
```

```
declare -A capitals=(
```

```
    [PL]="Warszawa"
```

```
    [DE]="Berlin"
```

```
    [FR]="Paryż"
```

```
)
```

```
echo "Stolica PL: ${capitals[PL]}"
```

Projekt współfinansowany ze środków Unii Europejskiej w ramach:
FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS)
"POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

133



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Declare - przykład 3

```
#!/usr/bin/env bash
```

```
# Zmienna eksportowana (-x)
```

```
declare -x PATH_SUFFIX="/my/tools/bin"
```

```
echo "PATH_SUFFIX = $PATH_SUFFIX"
```

```
# w podprocesie będzie widoczna jako zmienna środowiskowa
```

```
# Automatyczna zamiana na małe litery (-l)
```

```
declare -l lower="HeLLo WoRLd"
```

```
echo "lower = $lower"          # hello world
```

```
lower="cześć świecie"
```

```
echo "lower = $lower"          # cześć świecie
```

Declare - przykład 4

```
#!/usr/bin/env bash
```

```
# Automatyczna zamiana na wielkie litery (-u)
```

```
declare -u upper="HeLLo WoRLd"
```

```
echo "upper = $upper"      # HELLO WORLD
```

```
# Referencja do innej zmiennej (-n)
```

```
name="Rafał"
```

```
declare -n ref=name        # ref wskazuje na 'name'
```

```
echo "ref = $ref"          # Rafał
```

```
ref="Nowa wartość"
```

```
echo "name = $name"        # Nowa wartość
```

Declare - przykład 5

```
#!/usr/bin/env bash
```

```
# Zmienna globalna z wnętrza funkcji (-g)
```

```
make_global() {  
    declare -g GLOBAL_VAR="Jestem globalna"  
}  
make_global  
echo "GLOBAL_VAR = $GLOBAL_VAR"
```

```
# Wyświetlenie atrybutów i wartości (-p)
```

```
declare -p counter VERSION fruits capitals PATH_SUFFIX lower upper GLOBAL_VAR
```

136

Wyrażenia arytmetyczne 1

- Powłoka potrafi wykonywać działania arytmetyczne na liczbach całkowitych.
- Fragment otoczony $\$((i))$ jest interpretowany przez powłokę jako wyrażenie arytmetyczne.
- W wyrażeniach arytmetycznych zmienne nie muszą mieć poprzedzającego je znaku \$.
- Wyrażenia arytmetyczne są obliczane także wewnątrz cudzysłówów.
- Wyrażenia arytmetyczne w bash mają takie same reguły jak w C i Javie, czyli te same operatory, priorytety i kierunek łączenia.

137

Wyrażenia arytmetyczne 2

- ++ Zwiększenie o jeden (prefiksowe i postfiksowe)
- -- Zmniejszenie o jeden (prefiksowe i postfiksowe)
- + Plus (dodawanie)
- - Minus (odejmowanie)
- * Mnożenie
- / Dzielenie (z obcięciem części ułamkowej)
- % Reszta z dzielenia
- ** Potęgowanie
- << Przesunięcie bitowe w lewo
- >> Przesunięcie bitowe w prawo
- & Bitowe AND
- | Bitowe OR

138

Wyrażenia arytmetyczne 3

- $\sim$ Bitowe NOT (negacja bitowa)
- $!$ Logiczne NOT (negacja logiczna)
- $\wedge$ Bitowe XOR (wyłączne OR)
- $,$ Kolejne obliczanie wyrażeń (najpierw lewe, potem
prawe,
wynikiem jest wartość prawego)

Wyrażenia arytmetyczne 4

- $<$ Mniejsze niż
- $>$ Większe niż
- $<=$ Mniejsze lub równe
- $>=$ Większe lub równe
- $==$ Równe

Wyrażenia arytmetyczne 5

- **-gt** jest operatorem porównania liczb całkowitych używanym w kontekście **test/[]/[[]]** do sprawdzenia, czy jedna liczba jest większa od drugiej.
- **>** w kontekście arytmetyki pojawia się wewnątrz **((...))** jako standardowy operator „większe niż” znany z C/Javy i służy do porównywania wartości liczbowych.

141

Arytmetyczna pętla for

- Pętla arytmetyczna for w Bashu ma formę bardzo podobną do tej znanej z Javy i C:

```
for (( inicjalizacja ; warunek zakończenia ; aktualizacja ))
```

- Po nagłówku pętli następuje blok:

```
do  
    polecenia...  
done.
```

142

Tablice 1

- Zmienne przechowujące wiele wartości.
- Dwa typy: indeksowane i asocjacyjne.
- Brak ograniczenia rozmiaru (prawie, maksymalnie 599147937791 elementów).
- Elementy nie muszą być przypisane kolejno.
- Tablica indeksowana używa kolejnych numerów od zera poczynając dla wartości.
- Tablica asocjacyjna wykorzystuje unikalne klucze w postaci łańcuchów znakowych.

143

Dostęp do elementów w tablicy indeksowanej

- Pojedynczy element z indeksem wprost:
`echo "${owoce[1]}"`
- Pojedynczy element z indeksem odwrotnym (od końca):
`echo "${owoce[-1]}"`
- Wszystkie elementy:
`echo "${owoce[@]}"`
`echo "${owoce[#]}"`

145

Sprawdzenie długości

- Liczba elementów w tablicy:
`echo "${#owoce[@]}"`
- Sprawdzenie długości elementu o danym indeksie w tablicy:
`echo "${#owoce[1]}"`
- Lista wszystkich indeksów:
`echo "${!owoce[@]}"`

146

Wycinanie elementów tablic indeksowanych

- Pobranie n elementów od pozycji m:
`echo "${owoce[@]:m:n}"`
- Pobranie wszystkich elementów od pozycji m do końca tablicy:
`echo "${owoce[@]:m}"`
- Pobranie ostatnich n elementów:
`echo "${owoce[@]:-n}"`

147

Iteracja po elementach tablicy

- Przejście w pętli po wszystkich elementach w tablicy jest możliwe na dwa sposoby:
 - po wartościach z pominięciem indeksu,
 - po indeksie otrzymując zarówno indeks jak i wartość elementu.

Iteracja po wartościach

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

```
scripts=(*.sh)
```

```
if [[ ${#scripts[@]} -eq 0 ]]; then
```

```
    echo "    (brak plików .sh)"
```

```
else
```

```
    for f in "${scripts[@]"; do
```

```
        echo "    - $f ($(wc -l < "$f") linii)"
```

```
    done
```

```
fi
```

```
toor@Mikasa:~/scripts$ bash list.sh
- example.sh (11 linii)
- list.sh (9 linii)
- script.sh (0 linii)
```

Iteracja po indeksach

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

```
scripts=(*.sh)
```

```
if [[ ${#scripts[@]} -eq 0 ]]; then
```

```
    echo "    (brak plików .sh)"
```

```
else
```

```
    for ((i=0; i<${#scripts[@]}; i++)); do
```

```
        f="${scripts[$i]}"
```

```
        echo "    - $f $(wc -l < "$f") linii)"
```

```
    done
```

```
fi
```

```
toor@Mikasa:~/scripts$ bash list.sh
- example.sh (11 linii)
- list.sh (9 linii)
- script.sh (0 linii)
```

150

Tablice asocjacyjne

- Klucze tekstowe zamiast numerycznych indeksów.
- Tworzenie z obligatoryjnym użyciem **declare** i wskazaniem typu.

```
declare -A config
```

```
# Przypisanie wartości
```

```
config["host"]="localhost"
```

```
config["port"]="8080"
```

```
config["user"]="admin"
```

```
# Przypisanie w inicjalizacji
```

```
declare -A options=([debug]="true" [verbose]="false")
```

151

Dostęp do tablicy asocjacyjnej

- Do wartości:

```
echo "${config[host]}"
```

- Do wszystkich wartości:

```
echo "${config[@]}"
```

- Do wszystkich kluczy:

```
echo "${!config[@]}"
```

- Liczba elementów:

```
echo "${#config[@]}"
```

152

Iteracja po kluczach

```
#~/bin/env bash
```

```
declare -A config=([host]="localhost" [port]="8080" [user]="admin")
```

```
for key in "${!config[@]}; do  
    echo "$key = ${config[$key]}"  
done
```

```
toor@Mikasa:~/scripts$ ./script.sh  
port = 8080  
host = localhost  
user = admin
```

153

Operacje na plikach i potokach

- I/O w Bashu to sposób zarządzania przepływem danych między poleceniami, terminalem i plikami.
- Bash używa trzech podstawowych strumieni **stdin**, **stdout** i **stderr** oraz operatorów przekierowań i potoków, które umożliwiają automatyzację pracy i łączenie narzędzi w większe procesy.

Przypomnienie

Składnia	Typ	Opis
<code>cmd1 cmd2</code>	Potok	Przekazuje standardowe wyjście <code>cmd1</code> jako standardowe wejście do <code>cmd2</code> .
<code>> plik</code>	Przekierowanie wyjścia	Kieruje standardowe wyjście do pliku, nadpisując jego wcześniejszą zawartość.
<code>< plik</code>	Przekierowanie wejścia	Pobiera standardowe wejście z pliku zamiast z klawiatury (stdin).
<code>>> plik</code>	Dopisywanie do pliku	Kieruje standardowe wyjście do pliku, dopisując na końcu jeśli plik już istnieje.

Tabela. Typy przekierowań w Bash

155

Rozszerzone przekierowania

- Rozszerzone przekierowania w Bashu to zestaw mechanizmów, które wykraczają poza proste operacje **cmd > plik** czy **cmd < plik** i pozwalają precyzyjnie sterować wieloma strumieniami naraz.
- W praktyce oznacza to, między innymi, przekierowywanie konkretnych deskryptorów plików, łączenie **stdout** ze **stderr**, używanie zamian procesowych oraz bezpieczniejsze formy zapisu z użyciem opcji **noclobber**.

156

Opcja noclobber 1

- Opcja **noclobber** powłoki bash, która blokuje przypadkowe nadpisanie istniejącego pliku przez przekierowanie i kilka pokrewnych operatorów.
- Po jej włączeniu, polecenia typu **cmd > plik** zakończą się błędem, jeśli plik już istnieje.
- Działa dla operatorów **>**, **>&** i **<>**.
- Opcja dotyczy zwykłego nadpisania i nie zadziała dla przekierowania dopisującego.

157

Opcja noclobber 2

- Opcje można włączyć w bieżącej sesji:

set -o noclobber

- Opcję można wyłączyć w bieżącej sesji:

set +o noclobber

- Na stałe dla interaktywnej powłoki można ją włączyć poprzez dopisanie **set -o noclobber** do pliku **.bashrc** w katalogu domowym użytkownika.

158

Opcja noclobber 3

```
#~/bin/env bash
```

```
set -o noclobber
```

```
echo "Hello Bash!" > out.txt
```

```
cat out.txt
```

```
echo "Witaj Bash!" > out.txt
```

```
cat out.txt
```

```
toor@Mikasa:~/scripts$ ./script.sh
Hello Bash!
./script.sh: linia 7: out.txt: nie można nadpisać istniejącego pliku
Hello Bash!
```

159

Deskryptory plików 1

- Deskryptory plików to liczby całkowite zaczynające się od 0, które odnoszą się do konkretnych strumieni danych powiązanych z procesem.
- Deskryptory plików są abstrakcją jądra: każdy proces widzi je jako liczby całkowite (0, 1, 2, 3, ...), za którymi kryją się otwarte pliki, potoki, terminale czy gniazda sieciowe.

160

Deskryptory plików 2

- Przy starcie procesu zazwyczaj otwarte są trzy deskryptory: standardowe wejście (0), standardowe wyjście (1) oraz standardowe wyjście błędów (2).
- Gdy proces otwiera dodatkowe pliki do odczytu lub zapisu, otrzymują one kolejne wolne numery deskryptorów, począwszy od 3.

161

Deskryptory plików 3

- Najczęstsze użycie deskryptorów w bashu to przekierowanie standardowego wyjścia błędów do pliku, aby komunikaty nie przewijały się po ekranie (**cmd 2> p_err**).
- Aby zapisać zarówno standardowe wyjście, jak i błędy, można użyć dwóch osobnych przekierowań (**cmd > p_out 2> p_err**).

162

Deskryptory plików 4

- Dodatkowe pliki otwierane przez proces w skrypcie basha dostają kolejne numery deskryptorów, zaczynając od 3, np. **exec 3>log.txt** otwiera plik do zapisu na deskryptorze 3.
- Bash pozwala jawnie wskazywać deskryptory w przekierowaniach:
 - **n > plik** – przekierowanie wyjścia z deskryptora n do pliku,
 - **n < plik** – przekierowanie wejścia do deskryptora n z pliku,
 - gdy n jest pominięte w **>**, domyślnie używany jest 1 (stdout).

163

Deskryptory plików 5

- Można łączyć deskryptory:
 - **2 > &1** – kieruje stderr (2) w to samo miejsce, co stdout (1),
 - **1 > &3** – kieruje stdout tam, gdzie jest deskryptor 3, co umożliwia tymczasowe przełączanie wyjścia między różne cele.
- Można też całkowicie „wyciszyć” dany strumień, używając specjalnego pliku `/dev/null`:
 - **cmd > /dev/null** – ignoruj zwykłe wyjście,
 - **cmd 2> /dev/null** – ignoruj błędy,
 - **cmd > out.log 2> /dev/null** – loguj tylko komunikaty bez błędów.

164

Deskryptory plików 6

- Kluczem do wygodnej pracy z rozszerzonymi przekierowaniami jest traktowanie deskryptorów jako numerowanych przekierowań, którymi można dowolnie kierować dane między procesami, plikami i terminalem – a bash daje składnię do ich przełączania bez grzebania w niskopoziomowych wywołaniach systemowych.

165

Printf 1

- W Bashu **printf** działa jak jego odpowiednik w C i formatuje dane według łańcucha formatującego i wypisuje je na standardowe wyjście.
- W przeciwieństwie do **echo**, **printf** nie dodaje automatycznie znaku nowej linii i trzeba użyć `\n` w formacie jeżeli taka nowa linia jest oczekiwana.

166

Printf 2

- W Bashu **printf** działa jak jego odpowiednik w C i formatuje dane według łańcucha formatującego i wypisuje je na standardowe wyjście.
- Polecenie printf może wypisać napis podobnie jak echo:

printf "hello world"

- W przeciwieństwie do **echo**, **printf** nie dodaje automatycznie znaku nowej linii i trzeba użyć **\n** w formacie jeżeli taka nowa linia jest oczekiwana.

167

Printf 3

- **printf** ma dwie części, co pozwala w pełni kontrolować sposób wyświetlania informacji:

printf format-string [arguments]

- Pierwsza część (format-string) to łańcuch opisujący specyfikację formatu i zwykle podaje się go jako stały napis w cudzysłowie.
- Druga część ([arguments]) to lista argumentów (napisy, wartości zmiennych itp.), które są podstawiane w miejsce specyfikatorów formatu.

168

Printf 4

- Specyfikator formatu może być wielokrotnie użyty i, zazwyczaj, odpowiada liczbie wszystkich argumentów.
- Jeżeli liczba specyfikatorów formatu jest inna niż argumentów to:
 - jeśli jest więcej argumentów to specyfikator jest powtarzany,
 - jeśli oczekiwanych jest więcej argumentów niż podano to brakujące traktowane są jak 0 lub pusty string (w zależności od specyfikatora).
- Specyfikacja formatu zaczyna się od znaku %, po którym następuje znak określający oczekiwany typ (np. %s, %d, %f itd.).

169

Printf 5

Specyfikator formatu	Opis
%c	znak ASCII
%d %i	liczba całkowita
%e %E %f %g %G	liczba zmiennoprzecinkowa
%o	liczba całkowita ósemkowa bez znaku
%s	łańcuch znakowy
%u	liczba całkowita bez znaku
%x %X	liczba całkowita szesnastkowa bez znaku

Tabela: specyfikatory formatu printf

170

Printf 6

- Polecenie **printf** pozwala sterować szerokością i wyrównaniem pól wyjściowych.
- Ogólny schemat formatu z modyfikatorami wygląda tak:

%flags width.precision format-specifier

- Szerokość pola to wartość liczbowa określająca minimalną liczbę znaków przeznaczonych na dane.

171

Printf 7

- Domyślnie zawartość pola o zadanej szerokości jest wyrównana do prawej.
- Aby uzyskać wyrównanie do lewej, trzeba użyć flagi - (minus) przed szerokością, np. **%-20s**.
- Flaga - i podanie szerokości spowoduje, że **printf** wypisze napis wyrównany do lewej w polu o szerokości wskazanej liczbie znaków.
- Jeśli napis ma mniej niż wskazana liczba znaków, to pole zostanie uzupełnione spacjami.

172

Printf 8

#!/bin/env bash

printf "*%s**\n" \$1**

printf "*%10s**\n" \$1**

printf "*%-10s**\n" \$1**

```
toor@Mikasa:~/scripts$ ./script.sh Bash
**Bash**
**      Bash**
**Bash      **
```

173

Printf 9

- Modyfikator precyzji w **printf** (część **width.precision**) dla wartości dziesiętnych i zmiennoprzecinkowych określa liczbę cyfr po przecinku, jakie pojawią się w wyniku.
- Dla napisów precyzja określa maksymalną liczbę znaków, które zostaną wydrukowane z danego łańcucha.

174

Printf 10

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

```
printf "***%.3s**\n" $1  
printf "***%10.3s**\n" $1  
printf "***%-10.3s**\n" $1
```

```
toor@Mikasa:~/scripts$ ./script.sh Bash  
**Bas**  
**          Bas**  
**Bas          **
```

175

Printf 11

- Zachowanie precyzji dla różnych specyfikatorów jest zależne od typu:
 - **%d, %I, %o, %u, %x, %X**: oznacza minimalną liczbę cyfr do wypisania i jeśli wartość ma mniej cyfr, jest uzupełniana wiodącymi zerami, domyślna precyzja wynosi 1.
 - **%e, %E**: oznacza minimalną liczbę cyfr do wypisania i gdy wartość ma mniej cyfr, jest uzupełniana zerami po przecinku, domyślna precyzja wynosi 10, a precyzja równa 0 powoduje, że kropka dziesiętna nie jest wypisywana.
 - **%f**: liczba cyfr po prawej stronie kropki dziesiętnej.
 - **%g, %G**: maksymalna liczba cyfr znaczących.

176

Specyfikatory formatu zmiennoprzecinowego 1

- Pozwalają wskazać sposób wyświetlania liczby w określonej notacji.
- W przypadku **%e** i **%E** otrzymujemy notację naukową z różną wielkością znacznika wykładnika przy ogólnym formacie:

([-]d.precision[+-]dd)

177

Specyfikatory formatu zmiennoprzecinowego 2

- Formaty dynamiczne:
 - **%g**: używa konwersji **%e** lub **%f**, w zależności od tego, która reprezentacja jest krótsza, a następnie usuwa zbędne zera na końcu.
 - **%G**: działa analogicznie, ale korzysta z wersji **%E** lub **%F**, też wybiera najkrótszą postać, i usuwa końcowe zera.

Specyfikatory formatu zmiennoprzecinkowego 3

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

```
value=1234,56789
```

lub kropka, zależy od formatu
lokalnego!

```
printf "[%f]\n" $value
```

```
printf "[%10.2f]\n" $value
```

```
printf "[%10.4f]\n" $value
```

```
printf "[%8.2f]\n" $value
```

```
printf "[%e]\n" $value
```

```
printf "[%15.3e]\n" $value
```

```
printf "[%15.6e]\n" $value
```

```
printf "[%10.2e]\n" $value
```

```
toor@Mikasa:~/scripts$ ./script.sh
[1234,567890]
[ 1234,57]
[ 1234,5679]
[ 1234,57]
[1,234568e+03]
[      1,235e+03]
[ 1,234568e+03]
[ 1,23e+03]
```

179

Flagi printf

- Flagi w printf w Bashu to znaki umieszczane po %, które zmieniają sposób formatowania liczby lub napisu.
- Służą do sterowania wyrównaniem, wypełnieniem i zapisem znaku (np. plusa, spacji, prefiksów 0x).
- Typowe flagi to:
 - -: wyrównanie do lewej w zadanym polu szerokości (domyślnie jest do prawej).
 - 0: wypełnienie pola zerami zamiast spacjami (dla liczb).
 - +: pokazywanie zawsze znaku liczby.
 - spacja: poprzedza liczby dodatnie spacją (ujemne mają -).
 - #: „alternatywny format”, np. dopisuje 0x dla heksadecymalnych, wymusza kropkę dziesiętną itp.

180

Dodatkowe specyfikatory printf 1

- Poza standardowymi specyfikatorami printf udostępnia jeszcze dwa dodatkowe, specyfikatory używane w miejsce **%s**:
 - **%b**: powoduje interpretację sekwencji ucieczki w stylu echo w łańcuchu argumentu (rozwija znaki specjalne).
 - **%q**: wypisuje napis w takiej postaci, aby mógł zostać bezpiecznie użyty jako wejście powłoki (dodaje znaki ucieczki do znaków specjalnych).

181

Dodatkowe specyfikatory printf 2

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

```
printf "%s: %s\n" "Hello\tBash!"
```

```
printf "%b: %b\n" "Hello\tBash!"
```

```
printf "%s: %s\n" "Hello Bash!"
```

```
printf "%q: %q\n" "Hello Bash!"
```

```
toor@Mikasa:~/scripts$ ./script.sh
%s: Hello\tBash!
%b: Hello          Bash!
%s: Hello Bash!
%q: Hello\ Bash\!
```

182

Interakcja z użytkownikiem 1

- Polecenie **read** jest drugą stroną mechanizmu I/O dla napisów w powłoce.
- Pozwala wczytywać dane z wejścia do zmiennych powłoki.
- Jedna linia ze stdin jest pobierana i dzielona na słowa.
- Podstawowa składnia ma postać:

read var1 var2

183

Interakcja z użytkownikiem 2

- Linia wejściowa jest rozbijana na słowa wg znaków z zmiennej środowiskowej IFS (zwykle spacja, tabulacja lub nowa linia).
- Kolejne słowa są przypisywane do zmiennych (tu **var1**, **var2**).
- Jeśli jest więcej słów niż zmiennych, nadmiar trafia do ostatniej zmiennej.
- Gdy nie podasz żadnych nazw zmiennych, cała linia wejścia trafia do domyślnej zmiennej **REPLY**.

184

Interakcja z użytkownikiem 3

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

```
printf "Podaj słowo: "  
read  
echo $REPLY
```

```
printf "Podaj imię: "  
read imie  
echo "Cześć $imie!"
```

```
toor@Mikasa:~/scripts$ ./script.sh  
Podaj słowo: Bash!  
Bash!  
Podaj imię: Rafał  
Cześć Rafał!
```

185

Whiptail 1

- Umożliwia dodawanie do skryptów Bash prostych, tekstowych okien dialogowych, menu, komunikaty, formularze i paski postępu.
- Służy do tworzenia bardziej przyjaznych, interaktywnych skryptów niż zwykłe **read** i **echo**.
- Jest lżejszą alternatywą dla **dialog** i bywa domyślnie dostępny np. w systemach Debian.

186

Whiptail 2

- Pozwala na łatwe zrealizowanie:
 - Pobierania danych od użytkownika.
 - Tworzenia menu wyboru i konfiguratorów.
 - Pokazywania komunikatów, ostrzeżeń i potwierdzeń.
 - Wyświetlania postępu operacji.

187

Whiptail 3

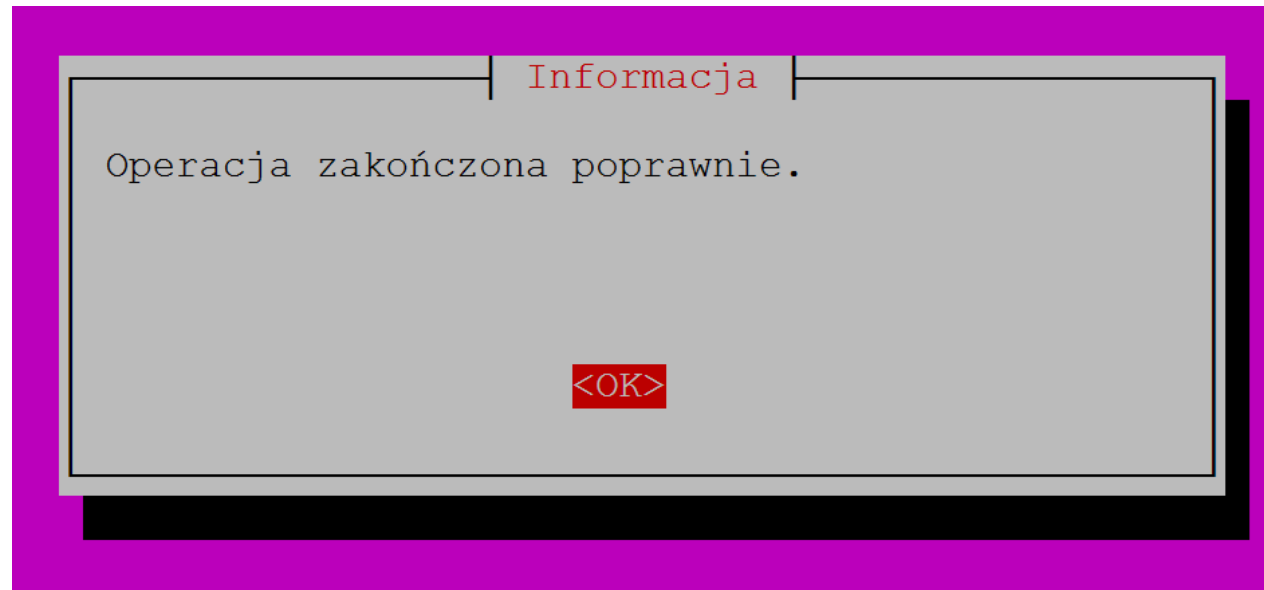
- Najważniejsze dostępne okna:

- **--msgbox** zwykły komunikat z przyciskiem OK.
- **--yesno** pytanie typu Tak/Nie.
- **--inputbox** pole do wpisania tekstu.
- **--passwordbox** pole do wpisania hasła.
- **--menu** menu wyboru jednej opcji.
- **--checklist** wybór wielu opcji.
- **--radiolist** wybór jednej opcji z listy.
- **--gauge** pasek postępu.

188

Whiptail 4

```
whiptail --title "Informacja" --msgbox "Operacja zakończona poprawnie." 10 50
```



189

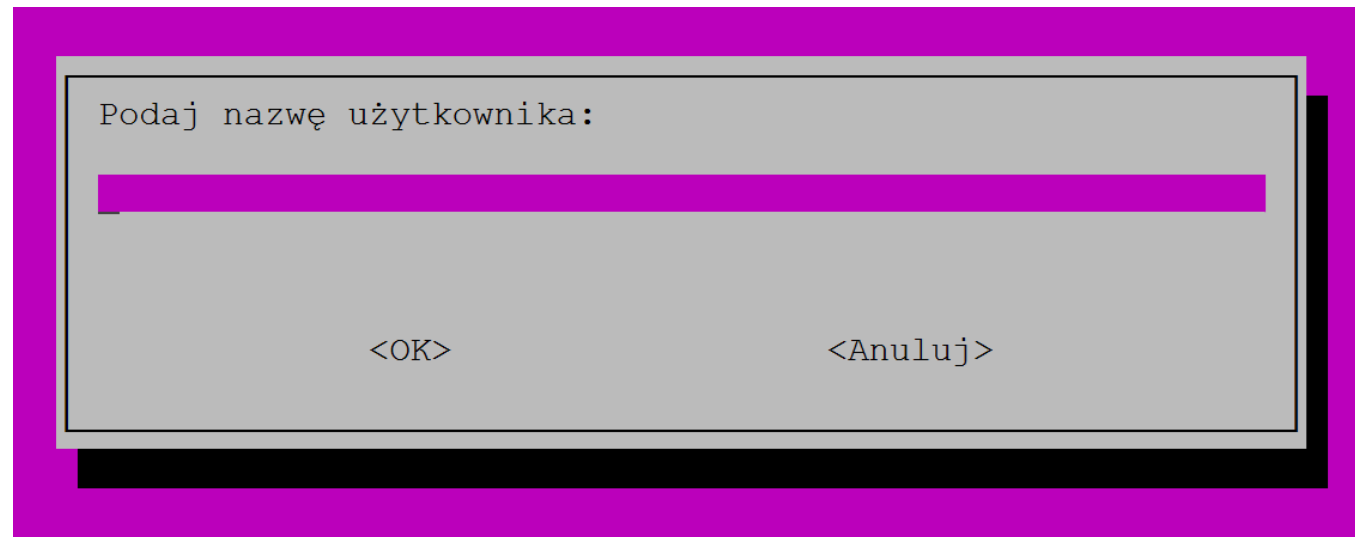
Whiptail 5

- Pobieranie danych w whiptail pozwala przechwycić wynik do zmiennej, ponieważ whiptail wypisuje wynik na **stderr**.
- W tym celu używa się kombinacji: **3 > &1 1 > &2 2 > &3**.

190

Whiptail 6

```
name=$(whiptail --inputbox "Podaj nazwę użytkownika:" 10 60 3 > &1 1 > &2 2 > &3)
```

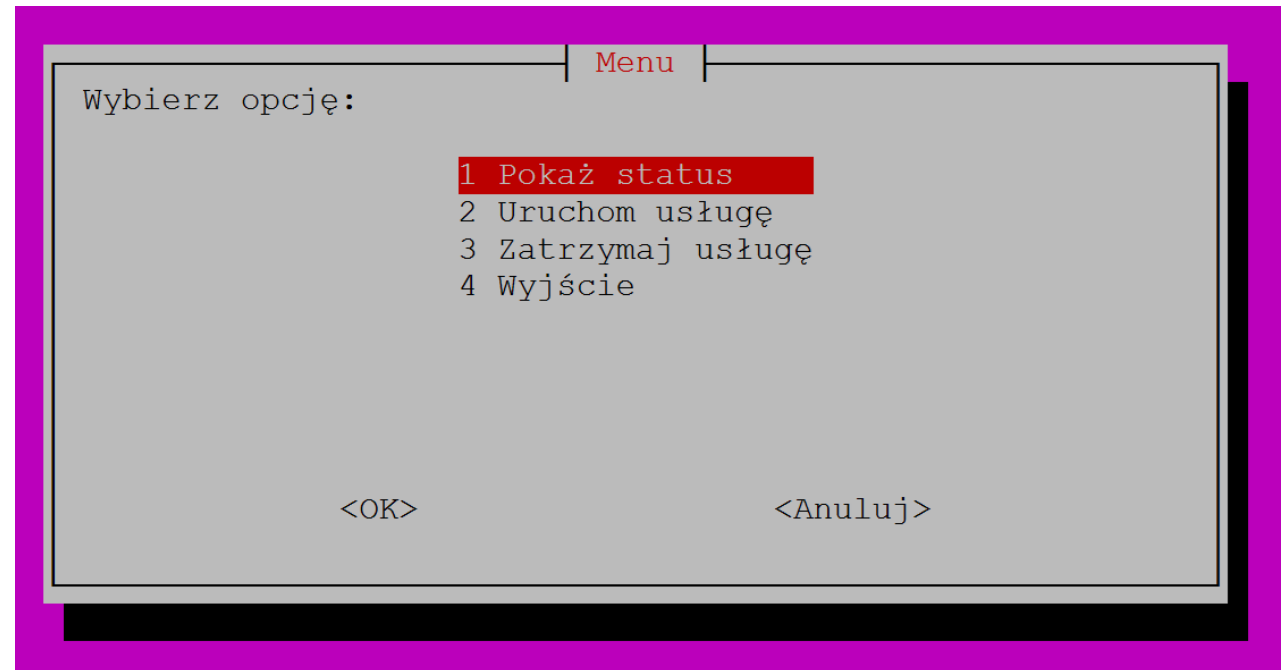


A screenshot of a Whiptail dialog box. The dialog box has a light gray background and a black border. It contains the text "Podaj nazwę użytkownika:" at the top. Below the text is a red rectangular input field. At the bottom of the dialog box, there are two buttons: "<OK>" on the left and "<Anuluj>" on the right. The entire dialog box is set against a solid red background.

191

Whiptail 7

```
choice=$(whiptail --title "Menu" --menu "Wybierz opcję:" 15 60 4 \  
"1" "Pokaż status" \  
"2" "Uruchom usługę" \  
"3" "Zatrzymaj usługę" \  
"4" "Wyjście" \  
3 > &1 1 > &2 2 > &3)
```



192

Whiptail 8

- Zalety: prostota, szybkość wdrożenia, małe wymagania, wygoda dla użytkownika terminala.
- Wady: mniej funkcji niż dialog, zależność od terminala, mniej wygodne przy bardzo rozbudowanych interfejsach.
- Whiptail najlepiej sprawdza się w instalatorach, konfiguratorach, menu administracyjnych i prostych narzędziach TUI.

193

Odczyt z pliku 1

- Do odczytu z pliku najczęściej używa się pętli **while** z **read**, gdzie wejście pętli jest przekierowane z pliku.
- Przygotowując odczyt konieczne jest wskazanie sposobu rozdzielania elementów (linie czy pojedyncze słowa) poprzez konfigurację **IFS**.
- Możliwe jest też uswienie interpretacji znaków ucieczki.

194

Odczyt z pliku 2

```
toor@Mikasa:~/scripts$ ./script.sh script.sh
Plik script.sh:
---
#!/bin/env bash

echo "Plik $1: "
echo "---"
while IFS= read -r line; do
    echo "$line"
done < $1
echo "---"
---
```

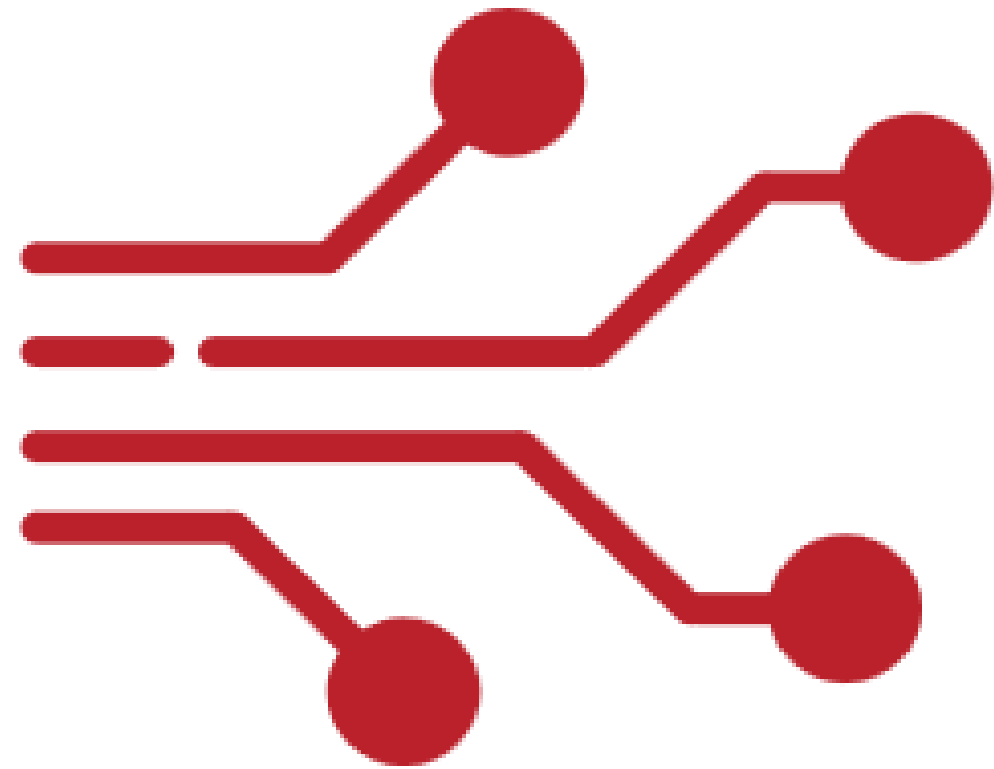
195

Automatyzacja zadań systemowych

- Bash jest podstawowym narzędziem administracyjnym w systemach Linux, bo pozwala łączyć komendy, pliki, harmonogram i usługi w proste skrypty automatyzujące pracę.
- Typowe zastosowania
 - Aktualizacja systemu i pakietów.
 - Archiwizacja i kopie zapasowe katalogów.
 - Monitorowanie użycia dysku, CPU i pamięci.
 - Czyszczenie starych logów i plików tymczasowych.
 - Automatyczne wysyłanie alertów i raportów administracyjnych.

196

Dobre praktyki



197

O czym warto pamiętać

- Zawsze należy zaczynać skrypt od poprawnego shebang, aby jednoznacznie wskazać interpreter.
- Warto włączyć tryb rygorystyczny **set -euo pipefail**, który pomaga szybciej wykrywać błędy wykonania i problemy z niezdefiniowanymi zmiennymi.
- Należy używać czytelnych nazw zmiennych i funkcji co pozwala, aby skrypt dało się utrzymywać także po kilku miesiącach.
- Dłuższy kod powinno podzielić się na funkcje, bo ułatwiają rozwój i utrzymanie skryptów.
- Wszystkie istotne operacje należy zapisywać do logu, co ułatwia analizę problemów i potwierdzenie wykonania zadania.

198

Najczęstsze błędy i jak ich unikać

- Należy uważać na spacje przy przypisaniach zmiennych, bo mają one znaczenie składniowe.
- Wskazane jest cytowanie zmiennych jako "**\$var**", co usuwa problem ze spacjami lub znakami specjalnymi.
- Należy weryfikować kody wyjścia poleceń co pozwoli przewidzieć błędne sytuacje ponieważ Bash opiera logikę warunków na exit status.
- Trzeba pamiętać, że return kończy funkcję, a exit kończy cały skrypt lub powłokę skryptową.
- Nie należy pisać dużych „jednoblokowych” skryptów.

199

Organizacja i styl kodu

- Warto stosować stały układ skryptu z wydzieloną konfiguracją, funkcjami pomocniczymi, częścią główną oraz obsługą błędów.
- Zmienne konfiguracyjne należy zgrupować początku pliku.
- Zachowywanie spójnych wcięć i odstępów pozwala, aby kod był czytelniejszy.
- Krótkie komentarze wyjaśniające „dlaczego”, są lepsze niż nie oczywiste „co”.
- Korzystaj z edytora z kolorowania składni, lintingiem i snippetami co przyspiesza wychwytywanie błędów przed uruchomieniem.

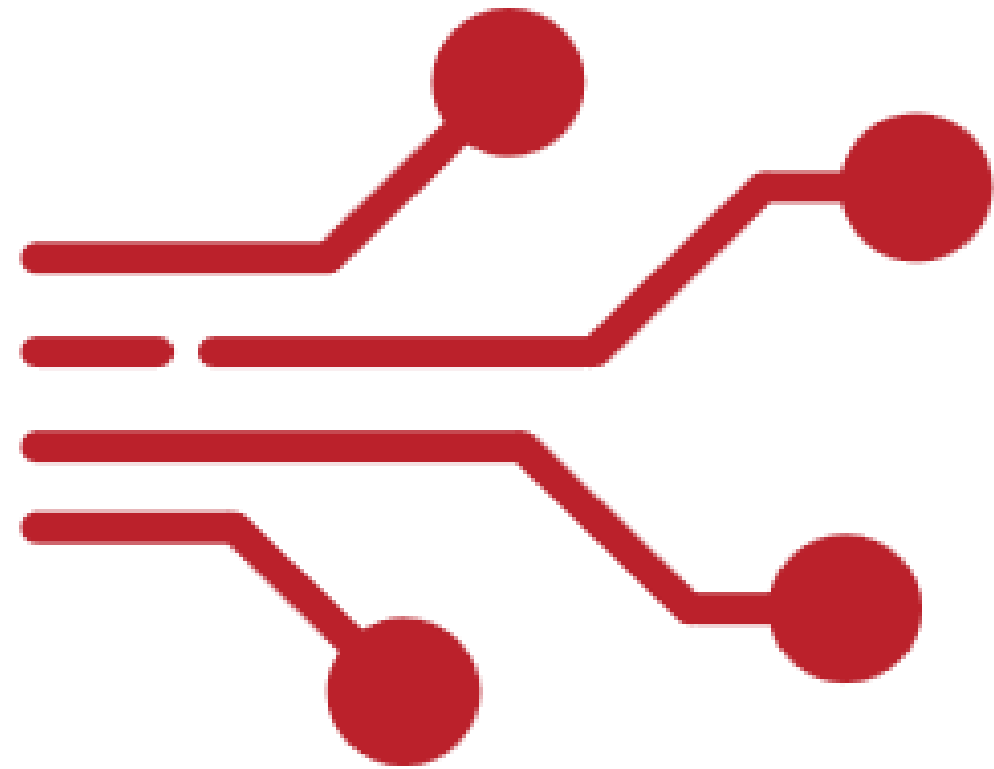
200

Bezpieczne praktyki

- Istnienie plików, katalogów i uprawnień należy przetestować przed wykonaniem operacji, używając mechanizmów testów warunków.
- Nadpisywanie lub usuwanie danych nie powinno odbywać się bez kontroli, a przy operacjach ryzykownych należy logować działanie i potwierdzać warunki wejściowe.
- Skryptów administracyjne muszą przewidywać sytuacje awaryjne, takie jak brak katalogu, brak miejsca, brak usługi, brak pliku wejściowego.
- Przy automatyzacji usług warto skorzystać z nadzoru systemd, bo dzięki temu system może monitorować proces i go restartować po awarii.

201

Debugowanie



202

Debugowanie skryptów Bash

- Debugowanie polega na obserwowaniu, które polecenia zostały wykonane, z jakimi danymi i z jakim wynikiem.
- Najprostsza technika to uruchomienie skryptu z **bash -x script.sh**, aby śledzić wykonywane linie poleceń.
- W samym skrypcie można czasowo włączyć **set -x**, aby uzyskać ślad wykonania tylko dla wybranego fragmentu.
- Do szybkiej diagnozy warto też wypisywać stan zmiennych za pomocą **echo** lub lepiej **printf**, który daje większą kontrolę nad formatem danych.

203

Praktyczne techniki debugowania

- Komunikaty diagnostyczne powinny być dodane przed i po kluczowych operacjach, takich jak, wejście do funkcji, kopiowanie pliku, wykonanie rsync, restart usługi.
- Warto wypisać wartości ważnych zmiennych, np. źródła, celu, nazwy pliku logu, aktualnej daty czy wyniku polecenia.
- Błędy i logi powinny być przekierowane do plików, aby później przeanalizować przebieg działania, bo deskryptory plików i przekierowania są do tego naturalnym mechanizmem.
- Należy ograniczać obszar problemu, czyli najpierw uruchamiać samą funkcję albo pojedyncze polecenie, zamiast cały skrypt.
- W przypadku gdy skrypt działa jako usługa, korzystać należy z systemctl status i logów usługi co przyspiesza znalezienie miejsca awarii.

204

Co sprawdzać podczas debugowania

- Należy zweryfikować, czy skrypt został uruchomiony właściwym sposobem, czyli bezpośrednio, przez interpreter, czy przez source, bo poszczególne tryby różnią się działaniem.
- Istnienie przed wykonaniem operacji wszystkie wymaganych plików, katalogów i uprawnień musi być zweryfikowane przed wykonaniem operacji.
- Konieczne jest sprawdzenie czy warunki w if zwracają oczekiwany exit status, ponieważ to on decyduje o prawdziwości warunku.
- Dane wejściowe powinny być skontrolowane czy nie zawierają pustych wartości, spacji albo nieoczekiwanych znaków specjalnych.
- Istotne jest też sprawdzenie czy problem dotyczy samego skryptu, czy zewnętrznego polecenia, np. tar, rsync, systemctl albo find.

205

Testowanie skryptów Bash

- Testowanie powinno obejmować nie tylko poprawne wykonanie, ale też sytuacje błędne i graniczne.
- Dla każdego skryptu warto sprawdzić co najmniej trzy scenariusze: dane poprawne, dane niepoprawne i brak wymaganych zasobów.
- Dobrą praktyką jest uruchamianie skryptu na bezpiecznych danych testowych, a nie od razu na środowisku produkcyjnym.
- W skryptach administracyjnych testuj zarówno sukces operacji, jak i reakcję na błąd, np. brak pliku, brak katalogu, niedostępny serwer lub zatrzymaną usługę.
- W materiałach o backupie szczególnie ważna jest nie tylko kopia, ale także test odtworzenia danych.

206

Minimalny zestaw testów

- Test uruchomienia: czy skrypt startuje bez błędu i poprawnie przyjmuje parametry wejściowe.
- Test warunków: czy if, case i testy plików reagują zgodnie z oczekiwaniem.
- Test błędów: czy skrypt kończy się czytelnie i zapisuje informację do logu, gdy wystąpi problem.
- Test danych skrajnych: pusty katalog, pusty plik, brak dopasowań, bardzo długa nazwa, brak uprawnień.
- Test wyniku: czy efekt końcowy naprawdę powstał, np. plik backupu istnieje, log został zapisany, usługa została uruchomiona ponownie.

207

Narzędzia wspierające jakość

- Linting pomaga wychwycić część błędów jeszcze przed uruchomieniem skryptu, a w prezentacji już pojawia się jako ważna funkcja edytora.
- Debugger w edytorze może uprościć analizę przepływu programu i stanu zmiennych.
- Logi są podstawowym narzędziem testowania i utrzymania, bo pozwalają potwierdzić, co naprawdę wydarzyło się podczas wykonania skryptu.
- printf bywa lepszy od echo przy diagnostyce, bo daje przewidywalny format wypisywania danych.
- Dobrze przygotowany zestaw małych danych testowych przyspiesza rozwój i ogranicza ryzyko błędów na produkcji.

208

Zalety i ograniczenia

- Zalety:
 - prostota,
 - szybkość tworzenia,
 - dostępność w systemie,
 - dobra integracja z narzędziami administracyjnymi.
- Ograniczenia:
 - trudniejsze utrzymanie dużych skryptów,
 - słabsza struktura niż w Pythonie czy Go,
 - ryzyko błędów przy słabej obsłudze wyjątków.
- Dlatego Bash najlepiej sprawdza się w małych i średnich automatyzacjach oraz jako warstwa orkiestrująca polecenia systemowe.

209

Obsługa usług 1

- W systemach z systemd podstawowym narzędziem do zarządzania usługami jest **systemctl**.
- Za jego pomocą można uruchamiać, zatrzymywać, restartować i sprawdzać stan usług, np. **systemctl start, stop, restart, status** oraz **is-active**.
- Skrypt Bash może sprawdzać, czy usługa działa, i automatycznie reagować na awarię.
- To prosty sposób na budowę mechanizmów self-healing dla usług systemowych.

210

Obsługa usług 2

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

```
if ! systemctl is-active --quiet nginx; then  
    echo "nginx nie działa; restart" >> /var/log/service-check.log  
    systemctl restart nginx  
fi
```

211

Harmonogram i usługi

- Skrypty Bash można uruchamiać cyklicznie przez **cron** albo przez timery **systemd**.
- Sam skrypt może też zostać uruchomiony jako usługa **systemd**, dzięki czemu system nadzoruje jego działanie i może go automatycznie restartować.

212

Skrypt jako usługa systemd 1

- Skrypty Bash zapisujący obciążenie systemu co minutę z nadanym prawem wykonywania:

```
#!/usr/bin/env bash
LOG=/var/log/myjob.log

while true; do
    # pobiera linię z uptime, np. "load average: 0.23, 0.18, 0.15"
    load=$(uptime | awk -F'load average:' '{print $2}' | sed 's/^ *//')
    echo "$(date '+%Y-%m-%d %H:%M:%S') load=${load}" >> "$LOG"
    sleep 60
done
```

213

Skrypt jako usługa systemd 2

- Plik konfiguracji usługi w `/etc/systemd/system`

```
[Unit]
Description=Skrypt bash jako usługa
After=network.target

[Service]
Type=simple
ExecStart=/usr/bin/env bash /usr/local/bin/myjob.sh
Restart=always
RestartSec=5
```

```
[Install]
WantedBy=multi-user.target
```

Skrypt jako usługa systemd 3

- Rejestracja i aktywacja usługi:

```
sudo systemctl daemon-reload      # wczytaj nowy unit
sudo systemctl enable myjob.service # start przy boot
sudo systemctl start myjob.service  # uruchom teraz
sudo systemctl status myjob.service # sprawdź status
```

215

Skrypt jako usługa systemd 4

```
root@Mikasa:/home/toor# systemctl status myjob.service
● myjob.service - Przykładowy skrypt bash jako usługa
   Loaded: loaded (/etc/systemd/system/myjob.service; disabled; preset: enabled)
   Active: active (running) since Tue 2026-04-07 14:34:45 CEST; 4s ago
     Main PID: 3877411 (bash)
       Tasks: 2 (limit: 9311)
      Memory: 824.0K
         CPU: 11ms
    CGroup: /system.slice/myjob.service
            └─3877411 bash /usr/local/bin/myjob.sh
              └─3877417 sleep 60

kwi 07 14:34:45 Mikasa systemd[1]: Started myjob.service - Przykładowy skrypt bash jako usługa.
```

216

Skrypt jako usługa systemd 5

```
root@Mikasa:/home/toor# cat /var/log/myjob.log
2026-04-07 14:34:45 load=0,33, 0,38, 0,35
2026-04-07 14:35:45 load=0,22, 0,34, 0,33
2026-04-07 14:36:45 load=0,62, 0,43, 0,37
2026-04-07 14:37:45 load=0,41, 0,42, 0,36
2026-04-07 14:38:45 load=0,15, 0,34, 0,34
```

217

Bash i backup 1

- Backup chroni przed utratą danych spowodowaną awarią dysku, błędem użytkownika, uszkodzeniem systemu lub atakiem ransomware.
- Ręczne wykonywanie kopii jest zawodne, dlatego backup powinien być zautomatyzowany.
- Skrypt Bash pozwala wykonywać kopie regularnie, przewidywalnie i z pełną kontrolą nad procesem.

218

Bash i backup 2

- Przyczyny wykorzystania skryptów Basha do backupów:
 - Są dostępne praktycznie na każdym systemie Linux/Unix.
 - Łatwo integrują się z narzędziami systemowymi i poleceniami administracyjnymi.
 - Dobrze nadają się do automatyzacji, harmonogramowania i logowania działań.
 - Sprawdzają się szczególnie w małych i średnich rozwiązaniach backupowych.

219

Bash i backup 3

- Typowe źródła backupów:
 - Katalogi użytkownika, np. /home, dokumenty, projekty, zdjęcia.
 - Pliki konfiguracyjne systemu, np. /etc.
 - Dane aplikacji, np. katalogi serwera WWW lub repozytoria.
 - Bazy danych, np. przez mysqldump lub inne narzędzia eksportu.

220

Bash i backup 4

- Główne podejścia do backupów z Bash:
 - Użycie **tar** tworzy archiwa pełnych kopii, co może być dobre do robienia prostych snapshotów i archiwizacji.
 - Użycie **rsync**, który kopiuje tylko różnice, nadaje się do backupów przyrostowych i synchronizacji katalogów.
- Polecenie **tar** pakuje wskazane katalogi do jednego archiwum.
- Często używa się go do backupu dziennego z nazwą zawierającą datę.
- To proste rozwiązanie, ale przy dużych danych bywa mniej wydajne niż **rsync**.

221

Bash i backup tar 2

```
#!/usr/bin/env bash
```

```
SRC="/home/toor/scripts"
```

```
DEST="/mnt/remote/backup"
```

```
DATE=$(date +%Y-%m-%d_%H-%M-%S)
```

```
mkdir -p "$DEST"
```

```
tar -czf "$DEST/backup_$(date +%Y-%m-%d_%H-%M-%S).tar.gz" "$SRC"
```

222

Bash i backup rsync 1

- Polecenie **rsync** potrafi ograniczyć kopiowanie tylko do zmienionych pliki, co oszczędza czas i miejsce.
- Obsługuje backup lokalny i zdalny przez SSH.
- Jest jednym z najczęściej używanych narzędzi do backupów w skryptach Bash.

223

Bash i backup rsync 2

```
#!/usr/bin/env bash
```

```
SRC="/home/toor/scripts/"
```

```
DEST="/mnt/remote/dane/"
```

```
rsync -av --delete "$SRC" "$DEST"
```

- Użyte opcje:
 - **-a** tryb archiwizacji; zachowuje strukturę, uprawnienia i daty plików.
 - **-v** tryb szczegółowy, pokazuje przebieg działania.
 - **--delete** usuwa z backupu pliki, które zostały usunięte ze źródła, dzięki czemu kopia jest lustrzana.

224

Bash i backup 5

- Backup powinien zapisywać przebieg do pliku logu.
- Log pozwala wykryć błędy, potwierdzić wykonanie zadania i analizować problemy.
- Może go realizować następujący kod:

```
LOG="/var/log/backup.log"
```

```
echo "$(date '+%F %T') Start backup" >> "$LOG"
```

225

Bash i backup 6

- Retencja i rotacja:
 - Backup bez retencji szybko zapełni dysk.
 - Dlatego zwykle trzyma się tylko ostatnie N kopii albo kopie z ostatnich X dni.
 - Starsze backupy można usuwać automatycznie przez **find ... -mtime** lub własną logikę rotacji.

```
find /backup -name "*.tar.gz" -type f -mtime +7 -delete
```

226

Bash i backup 7

- Weryfikacja i odtwarzanie:
 - Samo utworzenie backupu nie wystarczy dlatego trzeba okresowo sprawdzać, czy da się z niego odtworzyć dane.
 - Dobrą praktyką jest testowe odtwarzanie wybranych plików lub całego katalogu.
 - Backup ma wartość tylko wtedy, gdy proces przywracania działa poprawnie.

227

Bash i backup 8

```
#!/usr/bin/env bash
set -euo pipefail
SRC="/home/user/scripts"
DEST="/mnt/remote/backup"
LOG="/var/log/backup.log"
DATE=$(date +%Y-%m-%d_%H-%M-%S)
RETENTION_DAYS=7
mkdir -p "$DEST"
echo "$(date '+%F %T') Start backup" >> "$LOG"
tar -czf "$DEST/backup_${DATE}.tar.gz" "$SRC" >> "$LOG" 2>&1
find "$DEST" -name "backup_*.tar.gz" -type f -mtime +$RETENTION_DAYS -delete
echo "$(date '+%F %T') Backup zakończony poprawnie" >> "$LOG"
```

228

Bash i backup (na sterydach)

- Borg + Bash:
 - Dedupikacja blokowa, bo przy wielu snapshotach repo rośnie znacznie wolniej niż przy tar pełnym za każdym razem.
 - Wbudowane szyfrowanie i kompresja.
 - Retencja na poziomie borg prune (daily/weekly/monthly) zamiast własnej rotacji katalogów.

Borg

229

Co dalej?

- Ćwicz pisanie własnych skryptów automatyzujących codzienne zadania.
- Rozwijaj znajomość przetwarzania tekstu sięgając po sed, awk, wyrażenia regularne i potoki.
- Ucz się pisać skrypty bardziej odporne i czytelne.
- Poznaj zastosowania administracyjne
- Traktuj Bash jako narzędzie do orkiestracji, a przy większych projektach łącz go z Pythonem, Gitem, Dockerem i narzędziami DevOps.

230

Dziękuję za uwagę