Zmienne, parametry i argumenty

Poziom: podstawy **Dotyczy: bash 4+/5+** **Czas: ~20 min**

Zmienne w skrypcie to nie tylko `zmienna=wartość`. Ta strona zbiera to, czego skrypt potrzebuje ponad podstawy z rozdziału o cytowaniu: zasięg zmiennych, argumenty wywołania, wartości domyślne, podstawianie napisów i tablice.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Fiszka

TYLKO ODCZYT

Przypisanie zmiennej lokalnej

```
katalog_roboczy=/var/lib/NazwaUslugi
```

Bez spacji wokół `=` — spacja zamieniłaby to w wywołanie komendy o nazwie `katalog_roboczy` z argumentem `=/var/lib/...`. Zmienna istnieje tylko w bieżącej powłoce i w procesach, które ta powłoka uruchomi jako `source`, nie w zwykłych procesach potomnych.

ZMIENIA SYSTEM

Wyeksportowanie zmiennej do procesów potomnych: export

```
export ZMIENNA_SRODOWISKOWA=produkcja
```

Od tego momentu każdy proces uruchomiony z tej powłoki — także inny skrypt wywołany przez `./inny.sh` — dostaje tę zmienną w swoim środowisku. Bez `export` zmienna zostaje widoczna wyłącznie tam, gdzie ją ustawiliśmy.

export = ustaw zmienną środowiskową widoczną dla uruchamianych programów

TYLKO ODCZYT

Zmienna widoczna tylko wewnątrz funkcji: local

```
policz() {
    local wynik=$(( $1 + $2 ))
    echo "$wynik"
}
```

Bez `local` zmienna `wynik` byłaby zmienną globalną skryptu i nadpisałaby każdą istniejącą zmienną o tej samej nazwie poza funkcją. `local` działa tylko wewnątrz funkcji — na najwyższym poziomie skryptu nie ma odpowiednika.

TYLKO ODCZYT

Stała, której nie da się nadpisać: readonly

```
readonly MAKSYMALNY_ROZMIAR_MB=1024
```

Próba przypisania nowej wartości kończy się błędem powłoki, a nie cichym pominięciem. Używamy tego do stałych konfiguracyjnych na początku skryptu, żeby przypadkowe `MAKSYMALNY_ROZMIAR_MB=0` dalej w kodzie zostało wykryte od razu, a nie po godzinie szukania.

Przykład próby nadpisania:

```
bash: MAKSYMALNY_ROZMIAR_MB: readonly variable
```

TYLKO ODCZYT

Odczyt parametrów pozycyjnych: $1 … $9

```
echo "Pierwszy argument: $1, drugi: $2"
```

Parametry pozycyjne to argumenty, z jakimi skrypt (albo funkcja) został wywołany. `$0` to nazwa samego skryptu, nie pierwszy argument.

□□ TYLKO ODCZYT

Parametr o numerze dwucyfrowym: ${10}

```
echo "${10}"
```

Od dziesiątego argumentu wzwyż nawiasy klamrowe są obowiązkowe. `$10` bez nich to `$1` ze sklejonym literałem `0`, czyli pierwszy argument z dopisaną cyfrą — nie dziesiąty argument.

□□ TYLKO ODCZYT

Liczba argumentów: $#

```
echo "Otrzymano $# argumentów"
```

Zwraca liczbę parametrów pozycyjnych, przydatne do walidacji przed dalszym przetwarzaniem.

□□ ZMIENIA SYSTEM

Wszystkie argumenty jako osobne słowa: "$@"

```
kopiuj() { cp -v -- "$@"; }
```

`"$@"` w cudzysłowie rozwija się w tyle osobnych, w pełni zacytowanych argumentów, ile naprawdę otrzymaliśmy — jedyny zapis, który przekazuje je dalej bez zmian, nawet jeśli zawierają spacje. Mechanizm cytowania i różnicę względem `"$*"` tłumaczy strona Cudzysłowy, escapowanie i ekspansja; tutaj liczy się jedna zasada praktyczna: w skryptach piszemy prawie zawsze `"$@"`, nigdy gołe `$@` ani `$*`.

□□ TYLKO ODCZYT

Nazwa skryptu: $0

```
echo "Uruchomiono jako: $0"
```

Zawiera dokładnie to, co wpisaliśmy przy wywołaniu — `./skrypt.sh`, `/opt/skrypty/skrypt.sh` albo samo `skrypt.sh`, zależnie od tego, jak skrypt odpaliliśmy. Do wyciągnięcia samej nazwy pliku bez ścieżki służy `basename "$0"`.

TYLKO ODCZYT

Przesunięcie parametrów pozycyjnych: shift

```
while [ $# -gt 0 ]; do
    echo "przetwarzam: $1"
    shift
done
```

`shift` usuwa `$1`, a wszystkie kolejne parametry przesuwają się o jedną pozycję w dół — dawne `$2` staje się nowym `$1`. Klasyczny sposób na przejście przez listę argumentów bez znajomości ich liczby z góry.

TYLKO ODCZYT

Wartość domyślna, gdy zmienna jest pusta: ${ZMIENNA:-domyślna}

```
katalog="${1:-/var/tmp}"
```

Jeśli `$1` istnieje i nie jest puste, `katalog` dostaje jego wartość. W przeciwnym razie dostaje `/var/tmp`. Sama zmienna `$1` nie zostaje zmieniona — to podstawienie tworzy nową wartość w locie.

TYLKO ODCZYT

Przerwanie z komunikatem, gdy zmienna jest pusta: ${ZMIENNA:?komunikat}

```
: "${NAZWA_USLUGI:?zmienna NAZWA_USLUGI jest wymagana}"
```

Jeśli `NAZWA_USLUGI` jest pusta lub nieustawiona, powłoka wypisuje komunikat na stderr i kończy skrypt z kodem 1 — bez tego trzeba by pisać osobny `if`. Polecenie `:` to wbudowana komenda, która nic nie robi poza zwróceniem sukcesu — potrzebujemy jej tutaj tylko po to, żeby podstawienie miało gdzie się wykonać.

Przykład:

```
bash: NAZWA_USLUGI: zmienna NAZWA_USLUGI jest wymagana
```

TYLKO ODCZYT

Przypisanie wartości domyślnej na trwałe: ${ZMIENNA:=domyślna}

```
: "${LOG_DIR:=/var/log/NazwaUslugi}"
```

Podobnie jak `:-`, ale gdy zmienna jest pusta, `:=` **zapisuje** do niej wartość domyślną, a nie tylko podstawia ją tymczasowo. Po tej linii `$LOG_DIR` ma tę wartość już na stałe, do końca skryptu.

TYLKO ODCZYT

Usunięcie najkrótszego dopasowania z początku: ${ZMIENNA#wzorzec}

```
sciezka=/etc/nginx/nginx.conf
echo "${sciezka#*/}"
```

Odcina od lewej strony najkrótszy fragment pasujący do wzorca. Podwojony znak (`##`) odcina najdłuższe możliwe dopasowanie.

Przykład:

```
etc/nginx/nginx.conf
```

TYLKO ODCZYT

Usunięcie dopasowania z końca: ${ZMIENNA%wzorzec}

```
plik=raport-2026-09-25.tar.gz
echo "${plik%.tar.gz}"
```

Odcina od prawej strony. `%` odcina najkrótsze dopasowanie, `%%` najdłuższe — przydatne przy nazwach z wieloma kropkami.

Przykład:

```
raport-2026-09-25
```

🔒 TYLKO ODCZYT

Podmiana fragmentu napisu: ${ZMIENNA/stare/nowe}

```
host=app01.ad.zielona.example
echo "${host/ad.zielona.example/zielona.example}"
```

Podmienia **pierwsze** wystąpienie `stare` na `nowe`. Podwojony ukośnik na początku wzorca (`${host//e/E}`) podmienia wszystkie wystąpienia.

Przykład:

```
app01.zielona.example
```

🔒 TYLKO ODCZYT

Długość napisu: ${#ZMIENNA}

```
haslo="TymczasoweHaslo123"
echo "${#haslo}"
```

Zwraca liczbę znaków — przydatne do walidacji minimalnej długości hasła czy identyfikatora, zanim pójdzie dalej.

Przykład:

```
19
```

🔒 TYLKO ODCZYT

Tablica indeksowana

```
serwery=(cass01 cass02 cass03)
echo "${serwery[1]}"
```

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

Indeksowanie od zera. `${serwery[@]}` rozwija się we wszystkie elementy jako osobne słowa — dokładnie ten sam mechanizm co `"$@"`, dlatego w cudzysłowie: `"${serwery[@]}"`.

Przykład:

```
cass02
cass01 cass02 cass03
3
```

◫◫ TYLKO ODCZYT

Tablica asocjacyjna: declare -A

```
declare -A porty=( [http]=80 [https]=443 [ssh]=22 )
echo "${porty[https]}"
for klucz in "${!porty[@]}"; do
    echo "$klucz -> ${porty[$klucz]}"
done
```

Klucze są napisami, nie kolejnymi liczbami. `declare -A` jest obowiązkowe przed pierwszym użyciem — bez niego bash tworzy zwykłą tablicę indeksowaną i klucze tekstowe zamieniają się w indeks `0`. `"${!porty[@]}"` zwraca listę kluczy, nie wartości.

Przykład:

```
443
ssh -> 22
http -> 80
https -> 443
```

◫◫ **declare** = zadeklaruj zmienną powłoki z określonym typem, na przykład jako liczbę całkowitą albo tablicę

◫◫ ZMIENIA SYSTEM

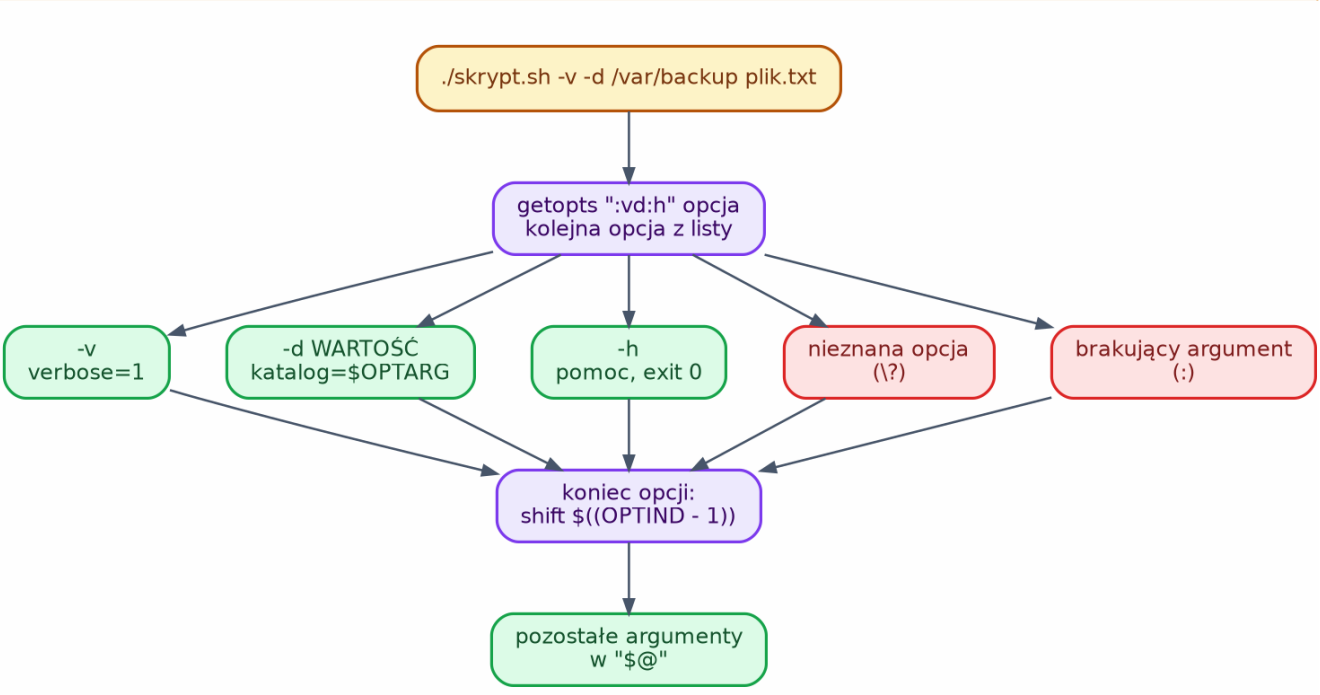
Parsowanie własnych opcji: getopts

```
#!/bin/bash
verbose=0
katalog=""

while getopts ":vd:h" opcja; do
    case "$opcja" in
        v) verbose=1 ;;
        d) katalog="$OPTARG" ;;
        h) echo "Użycie: $0 [-v] [-d KATALOG]"; exit 0 ;;
        \?) echo "Nieznana opcja: -$OPTARG" >&2; exit 2 ;;
        :) echo "Opcja -$OPTARG wymaga argumentu" >&2; exit 2 ;;
    esac
done
shift $((OPTIND - 1))

echo "verbose=$verbose katalog=$katalog pozostale=$*"
```

`getopts` czyta opcje w stylu `-v`, `-d wartosc` zgodnie z definicją w pierwszym argumencie (`:vd:h`) — dwukropek po literze oznacza, że opcja wymaga argumentu, wiodący dwukropek włącza własną obsługę błędów zamiast komunikatów `getopts`. `$OPTIND` po pętli wskazuje pierwszy nieprzetworzony argument, `shift $((OPTIND - 1))` usuwa już rozpoznane opcje, zostawiając w `$@` argumenty pozycyjne.



Getopts w pętli rozpoznaje kolejną opcję, rozgałęzia się według litery albo błędu, a po ostatniej opcji shift zostawia resztę argumentów w "$@" (kliknięcie otwiera pełny rozmiar)

Przykład wywołania `./skrypt.sh -v -d /var/backup plik.txt`:

```
verbose=1 katalog=/var/backup pozostale=plik.txt
```

TYLKO ODCZYT

Wszystkie podstawienia parametrów

Zapis	Działanie
`${ZMIENNA:-domyslna}`	podstawia domyślną, jeśli zmienna pusta/nieustawiona, nie zapisuje jej
`${ZMIENNA:=domyslna}`	jak wyżej, ale zapisuje domyślną do zmiennej
`${ZMIENNA:?komunikat}`	przerywa skrypt z komunikatem, jeśli zmienna pusta/nieustawiona
`${ZMIENNA:+wartosc}`	podstawia `wartosc`, jeśli zmienna JEST ustawiona (odwrotność `:-`)
`${ZMIENNA#wzorzec}`	usuwa najkrótsze dopasowanie od początku
`${ZMIENNA##wzorzec}`	usuwa najdłuższe dopasowanie od początku
`${ZMIENNA%wzorzec}`	usuwa najkrótsze dopasowanie od końca
`${ZMIENNA%%wzorzec}`	usuwa najdłuższe dopasowanie od końca
`${ZMIENNA/stare/nowe}`	podmienia pierwsze wystąpienie
`${ZMIENNA//stare/nowe}`	podmienia wszystkie wystąpienia
`${#ZMIENNA}`	długość napisu w znakach

Uważaj

- `getopts` obsługuje tylko krótkie opcje jednoliterowe — długie opcje (`--verbose`) wymagają ręcznego `case` na `$1` i `shift`.
- Tablica asocjacyjna bez wcześniejszego `declare -A` nie zgłosi błędu — po cichu zachowa się jak zwykła tablica indeksowana, co jest trudne do zdiagnozowania.
- `local zmienna=$(komenda)` maskuje kod wyjścia `komenda` kodem wyjścia `local` — omawiamy to jako osobną pułapkę na stronie Tryb ścisły: set -euo pipefail.

Z życia wzięte

Z ŻYCIA WZIĘTE

Środa, 10:00: Ola pisze skrypt wdrożeniowy dla zespołu i przekazuje ścieżkę katalogu przez `$10`, licząc na dziesiąty argument z listy plików do spakowania. Skrypt zawsze dokleja `0` do pierwszego argumentu zamiast czytać dziesiąty, bo `$10` to `$1` sklejone z cyfrą `0`, nie prawdziwy odczyt dwucyfrowego numeru. Dopiero zapis `${10}` z nawiasami klamrowymi naprawia wynik na cass01, gdzie skrypt miał zbierać telemetrię z dziesięciu wózków naraz.

Po co to istnieje

Skrypt bez parametrów i bez zmiennych roboczych to skrypt, który robi dokładnie jedną rzecz dla dokładnie jednego przypadku. W praktyce chcemy ten sam skrypt backupu uruchomić dla trzech różnych baz, ten sam skrypt czyszczący z inną liczbą dni retencji w zależności od środowiska, tę samą procedurę wdrożeniową z opcją `--dry-run` do testu na sucho. Parametry, zmienne o kontrolowanym zasięgu i podstawienia napisów to narzędzia, które zamieniają skrypt jednorazowy w skrypt wielokrotnego użytku.

Jak to działa

Rozwiń: jak to działa pod spodem

Pojęcie: zasięg zmiennej. Zakres kodu, w którym dana zmienna jest widoczna i modyfikowalna. W bashu domyślny zasięg jest globalny dla całego skryptu — zmienna ustawiona w jednej funkcji jest widoczna w każdej innej, chyba że jawnie ograniczymy ją słowem `local`. To jest zachowanie odwrotne niż w większości języków programowania, gdzie domyślnym zasięgiem zmiennej wewnątrz funkcji jest zasięg lokalny.

```
licznik=0


zwieksz() {
    licznik=$((licznik + 1))
}
```

```
zwieksz
zwieksz
echo "$licznik"
```

```
2
```

Gdyby `licznik` w funkcji był poprzedzony `local`, funkcja tworzyłaby własną, oddzielną kopię przy każdym wywołaniu, a globalny `licznik` pozostałby na zero. Świadomy wybór między zmienną globalną a `local` to jeden z najczęstszych źródeł błędów w dłuższych skryptach z wieloma funkcjami — stąd zasada, którą wprowadzamy na stronie Funkcje: każda zmienna pomocnicza wewnątrz funkcji dostaje `local`, chyba że celowo ma modyfikować stan globalny.

`export` a zasięg między procesami. `local` i zmienna globalna dotyczą jednej powłoki. `export` to zupełnie inny wymiar — decyduje, czy zmienna trafia do **środowiska**, czyli listy par klucz-wartość kopiowanej do każdego procesu potomnego. Zmienna nieeksportowana istnieje tylko w pamięci bieżącej powłoki; zmienna eksportowana zostaje skopiowana do każdego programu, jaki ta powłoka uruchomi — w tym do innych skryptów wywołanych przez `./inny.sh`, ale nie z powrotem do rodzica, jeśli proces potomny ją zmieni.

Parametry pozycyjne nie są tablicą. `$1`, `$2` i tak dalej wyglądają jak elementy tablicy, ale są osobnym mechanizmem powłoki — nie da się ich przypisać pojedynczo (`$1=wartosc` to błąd składni), zmienia się je tylko przez `shift` albo przez `set -- nowa lista`. `$@` i `$*` to dwa różne sposoby odczytu tej samej listy naraz, opisane szczegółowo na stronie o cytowaniu — reguła dla skryptów jest jedna: piszemy `"$@"`, gdy chcemy przekazać argumenty dalej bez zmian.

Krok po kroku

1. Definiujemy skrypt z wartością domyślną i walidacją wymaganego argumentu.

```
#!/bin/bash
srodowisko="${1:-dev}"
: "${BACKUP_TARGET:?ustaw zmienną BACKUP_TARGET przed uruchomieniem}"
echo "Wdrażamy na środowisko: $srodowisko, cel backupu: $BACKUP_TARGET"
```

2. Uruchamiamy bez wymaganej zmiennej środowiskowej.

```
./wdroz.sh produkcja
```

```
bash: BACKUP_TARGET: ustaw zmienną BACKUP_TARGET przed uruchomieniem
```

3. Uruchamiamy poprawnie.

```
BACKUP_TARGET=/mnt/backup ./wdroz.sh produkcja
```

```
Wdrażamy na środowisko: produkcja, cel backupu: /mnt/backup
```

Częste błędy

Brak `local` w funkcji nadpisuje zmienną globalną.

```
plik="/etc/hosts"

sprawdz() {
    for plik in *.log; do
        echo "$plik"
    done
}

sprawdz
echo "$plik"
```

Po wywołaniu `sprawdz` zmienna `plik` w reszcie skryptu wskazuje na ostatni plik `.log` z pętli, a nie na `/etc/hosts`. Naprawa: `local plik` na początku funkcji.

`local zmienna=$(komenda)` gubi kod wyjścia komendy.

```
pobierz_status() {
    local status=$(systemctl is-active nginx)
    echo "$status"
}
```

`local` jest tu komendą samą w sobie, więc `$?` po tej linii to kod wyjścia `local` (prawie zawsze 0), nie kod wyjścia `systemctl`. Rozdzielenie deklaracji i przypisania naprawia problem:

```
pobierz_status() {
    local status
    status=$(systemctl is-active nginx)
    echo "$status"
}
```

Mylenie `$*` z `"$@"` przy przekazywaniu argumentów dalej.

```
uruchom_wszystko() {
    dla_kazdego_skryptu.sh $*
}
```

Argument z spacją rozpada się na kilka, dokładnie tak jak przy niecytowanej zmiennej. Poprawnie: `dla_kazdego_skryptu.sh "$@"`.

`${10}` zapisane jako `$10`.

```
echo "Dziesiąty argument: $10"
```

Bash odczytuje to jako `$1` sklejone z literalną cyfrą `0`. Przy wywołaniu z argumentem `raport` na pierwszej pozycji dostaniemy `raport0`, a nie dziesiąty parametr. Naprawa: nawiasy klamrowe, `${10}`.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — shift i zliczanie argumentów

```
cd /tmp
cat > lista.sh <<'EOF'
#!/bin/bash
echo "Otrzymano $# argumentów"
while [ $# -gt 0 ]; do
    echo " - $1"
    shift
done
EOF
chmod +x lista.sh
./lista.sh "pierwszy plik.txt" drugi trzeci
```

Ile linii z myślnikiem zobaczymy i dlaczego pierwszy argument nie rozpadł się na dwa?

Odpowiedź

Trzy linie — po jednej na argument, mimo że pierwszy zawiera spację. `"$1"` w cudzysłowie chroni go przed podziałem na słowa, tak samo jak każdą inną zmienną. Gdyby pętla używała `echo $1` bez cudzysłowów, wynik byłby ten sam tylko dlatego, że `echo` sam zlepia argumenty spacją przy wypisywaniu — prawdziwy test to `printf '[%s]\n' "$1"`.

Ćwiczenie 2 — tablica asocjacyjna portów

```
cd /tmp
declare -A uslugi=( [www]=80 [ssh]=22 [db]=5432 )
for nazwa in "${!uslugi[@]}"; do
    printf '%-5s -> %s\n' "$nazwa" "${uslugi[$nazwa]}"
done
```

Odpowiedź

Kolejność iteracji po kluczach tablicy asocjacyjnej nie jest gwarantowana — bash nie sortuje jej alfabetycznie ani według kolejności wstawienia. Jeśli kolejność ma znaczenie w wyniku, sortujemy klucze jawnie: `for nazwa in $(printf '%s\n' "${!uslugi[@]}" | sort); do`.

Pytania kontrolne

1. Dlaczego `${ZMIENNA:-domyslna}` nie zmienia wartości `ZMIENNA`, a `${ZMIENNA:=domyslna}` tak?

Odpowiedź

`:-` tylko podstawia wartość domyślną w miejscu tego konkretnego wyrażenia — sama zmienna zostaje bez zmian, co widać, gdy odczytamy ją zaraz potem. `:=` dodatkowo wykonuje przypisanie, więc po jego użyciu zmienna ma tę wartość trwale, aż do końca skryptu albo kolejnego przypisania.

2. Skrypt ma sygnaturę `moj_skrypt.sh -v plik1 plik2`. Co zwróci `$OPTIND` po pętli `getopts ":v" opcja`, i do czego to służy?

Odpowiedź

`$OPTIND` wskaże `2` — indeks pierwszego argumentu, który nie jest opcją (tu: `plik1`, licząc od 1). Używamy `shift $((OPTIND - 1))`, żeby usunąć z listy parametrów pozycyjnych wszystko, co `getopts` już rozpoznał jako opcje, i zostać z samymi argumentami do dalszego przetwarzania w `"$@"`.

Powiązane: Cudzysłowy, escapowanie i ekspansja · Pierwszy skrypt: shebang, uprawnienia, uruchomienie · Funkcje · Zmienne środowiskowe i PATH

Źródła: `man 1 bash` (sekcje *Parameters*, *Arrays*, *SHELL BUILTIN COMMANDS* → `getopts`) · `help getopts` · POSIX.1-2024

Revision #11
Created 2026-09-25 00:40:45 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:15 UTC by Tomasz Konieczny