RÓŻNE DZIWNE SZCZEGÓŁY

Różne dziwne szczegóły

```
# otwarcie uchwytu do pisania do potoku
# open(HANDLE,"|pipe");

open HANDLE,"|more";
for($i=0; $i<100; $i++) {
   print HANDLE "*"x60 . "\n";
}
close HANDLE;
```

Różne dziwne szczegóły

```
# otwarcie uchwytu do czytania z potoku
# open(HANDLE,"pipe|");

open HANDLE,"ls / -lR |";
@allfiles = <HANDLE>;
close HANDLE;
foreach $file (sort @allfiles) {
    print "$file\n";
}
```

Różne dziwne szczegóły

```
# operator wieloznacznej nazwy pliku
# <nazwa_wieloznaczna>
# patrz również → funkcja glob()

$skalar = <*.c>
@tablica = <*.html>

while($name = <*.c>) {
   chmod 0644,$name;
}
foreach $name (<*.c>) {
   chmod 0644,$name;
}
```

Różne dziwne szczegóły

```
# operator zakresu (n..m)

@tablica = (1..10);

foreach $i (1..10) {
   print "$i\n";
}

@alfabet = ('A'..'Z');
@dni_miesiaca = ('01'..'31');
@kombinacje = ('aa'..'zz');

$halfbyte = 11
$hexdig = ('0'..'9','A'..'F')[$halfbyte];
```

Różne dziwne szczegóły

```
# operator przełącznika dwustanowego
# on_expression .. off_expression



#wychwytywanie załącznika uu
while($line = <HANDLE>) {
  if($line =~ /^begin / .. $line =~ /^end/) {
     print $line;
  }
}
```

Różne dziwne szczegóły

```
# zmienna $.
# numer aktualnej linii w ostatnio czytanym
# uchwycie wejściowym

# wypisz linie od 100 do 110

open(HANDLE,"/etc/passwd");
while($line = <HANDLE>) {
  if($. == 100 .. $. == 110) {
     print "$line\n";
  }
}
```

Różne dziwne szczegóły

```
# uwaga! jeśli operator przełącznika
# dwustanowego zawiera literały całkowite
# TO uznaje się, że są one porównywane
# ze zmienną $.

# wypisz linie od 100 do 110

open(HANDLE,"/etc/passwd");
while($line = <HANDLE>) {
  if(100 .. 110) {
     print "$line\n";
  }
}
```

Różne dziwne szczegóły

```
# zmienna $$
# PID procesu Perla interpretującego skrypt

print "My PID = $$\n";
```

Różne dziwne szczegóły

```
# zmienna $[
# dolna granica indeksów tablic

# dla miłośników Fortranu

$[ = 1;
for $i (1..10) {
   $t[$i] = $i;
}
```

Różne dziwne szczegóły

```
# zmienna $]
# wersja interpretera Perla

print "Coś takiego, masz perla w wersji $]\n";
```

Różne dziwne szczegóły

```
# zmienna $O
# nazwa systemu operacyjnego

print "No pięknie, masz $O\n";
```

Różne dziwne szczegóły

```
# dokument „osadzony”

$var = 111;

print <<EOF;
to jest $var
EOF

print <<”EOF”;
to jest $var
EOF

print <<’EOF’;
echo $var
EOF
```

Różne dziwne szczegóły

```
# <STDIN>, $_ i while

while (defined($_ = <STDIN>)) { print $_; }
while (<STDIN>) { print; }
for( ; <STDIN> ; ) { print; }
print $_ while defined ($_ = <STDIN>);
print while <STDIN>;

# ŹLE!
if (<STDIN>) { print; }

# DOBRZE!
if ($_ = <STDIN>) { print; }
```

Różne dziwne szczegóły

```
# co to jest <> ?

while (<>) { ... }

@ARGV = (’-’) unless @ARGV;
while($ARGV = shift) {
    open(ARGV,$ARGV) or warn ”$ARGV???\n”;
    while(<ARGV>) {
       .....
    }
}

# uwaga! Wolno zmodyfikować @ARGV przed
# użyciem <>
```

Zalecenia stylistyczne

„I have my reasons for each of these things, but I don't claim that everyone else's mind works the same as mine does.”

Larry Wall

Stosuj 4-kolumnowe wcięcia

```
#!/usr/bin/perl

$i=5;
while($i > 0) {
____print ”hello world\n”;
____$i--;
____if(i == 0) { print ”to koniec”; }
}
```

Stawiaj { w tej samej linii co słowo kluczowe, albo wyrównuj { do tej samej kolumny

```
#!/usr/bin/perl

$i=5;
while($i > 0) {
    print ”hello world\n”;
    $i--;
    if(i == 0) { print ”to koniec”; }
}
```

Bloki jednoliniowe zapisuj w jednej linii

```
#!/usr/bin/perl

$i=5;
while($i > 0) {
    print ”hello world\n”;
    $i--;
    if(i == 0) { print ”to koniec”; }
}
```

Nie stawiaj spacji przed średnikami

```
#!/usr/bin/perl

$i=5;
while($i > 0) {
    print ”hello world\n”;
    $i--;
    if(i == 0) { print ”to koniec”; }
}
```

Otaczaj operatory spacjami

```
#!/usr/bin/perl

$i=5;
while($i > 0) {
    print ”hello world\n”;
    $i--;
    if(i == 0) { print ”to koniec”; }
}
```

Otaczaj spacjami złożone wyrażenia indeksujące

```
#!/usr/bin/perl

$tab1[$i] = tab2[ 2 * $i - 1 ]
```

else stawiaj w nowym wierszu

```
#!/usr/bin/perl

if($i) { $i **= 2;}
else { $i = 2; }
```

Wstawiaj puste linie pomiędzy fragmentami kodu o różnym przeznaczeniu

```
#!/usr/bin/perl

for($i=0; $i < 100; $i++) { $tab1[$i] = $i; }
for($i=0; $i < 100; $i++) { $tab2[$i] = 0; }

open(FILE, ”<file”);
@lines = <FILE>;
close FILE;
```

Nie umieszczaj spacji pomiędzy nazwą funkcji a nawiasem otwierającym listę argumentów

```
#!/usr/bin/perl

open(FILE, ”<file”);
@lines = <FILE>;
close(FILE);
```

Wstawiaj spację po każdym przecinku

```
#!/usr/bin/perl

open(FILE, ”<file”);
@lines = <FILE>;
close(FILE);
```

Używaj **eval**, kiedy nie jesteś pewien, że coś musi się udać

```
eval WYRAŻENIE
eval BLOK

zwraca taką wartość, jak WYRAŻENIE lub BLOK albo
undef

zmienna $@ będzie zawierać komunikat błędu

najprostszy interpreter Perla w Perlu:

while(<>){ eval; print $@; }

wychwycenie dzielenia przez zero:

eval { $x = $y / $z; }; warn $@ if $@;
```

Sprzątanie po programie

```
#! /usr/bin/perl

eval <<ENDOFPROGRAM
#
# Twój kod
#
ENDOFPROGRAM
# sprzątanie
unlink ”/tmp/tmp/tmpfile”;
```

Łam długie linie po operatorze ALE nie po **or** albo **and**

```
#!/usr/bin/perl

$wyznacznik = $wspolczynnik_B ** 2 -
 4 * $wspolczynnik_A * $wspolczynnik_C;

open(FILE, ”<file”) or die ”Plik znikł”;
@lines = <FILE>;
close(FILE);
```

Wstawiaj spację po nawiasie zamykającym, jeśli jego nawias otwierający leży w innej linii

```
#!/usr/bin/perl

$var = fun(fun(fun(fun(fun(fun(fun(fun(2))))))
) ) ) ) );
```

Wyrównuj w pionie odpowiadające sobie elementy

```
#!/usr/bin/perl

my $i          = 0;
my $zmienna    = 100;

mkdir $tmpdir, 0700 or die "can't mkdir $!";
chdir($tmpdir)      or die "can't chdir $!";
mkdir 'tmp',   0777 or die "can't mkdir $!";
```

Pomijaj zbędne ograniczniki, jeśli nie cierpi na tym czytelność

```
#!/usr/bin/perl

open FILE, ”>new”;
close FILE;
```

To, że **MOŻESZ** zrobić coś w pewien sposób, nie znaczy, że **MUSISZ** tak to zrobić, na przykład:

```
open(FOO,$foo) || die "Can't open $foo";
```

jest lepsze niż:

```
die "Can't open $foo" unless open(FOO,$foo);
```

Jeśli **MOŻESZ** opuścić nawiasy, to nie zawsze **MUSISZ** to zrobić, na przykład:

```
return print reverse sort values %array;
```

wygląda gorzej niż:

```
return print(reverse(sort (values(%array))));
```

Myśl życzliwie o osobie, która przejmie Twój kod - jeśli zacznie sama wstawiać nawiasy, z pewnością zrobi to źle.

Larry Wall

Oprócz wcięć stosuj wycięcia, szczególnie dla instrukcji **last**

```
LINE:
    for (;;) {
        statements;
      last LINE if $foo;
        next LINE if /^#/;
        statements;
    }
```

Stosuj etykiety pętli - to zwiększa czytelność

```
LINE:
    for (;;) {
        statements;
      last LINE if $foo;
        next LINE if /^#/;
        statements;
    }
```

Nie używaj odwróconych apostrofów, jeśli chcesz odrzucić zwracaną przez nie wartość - użyj funkcji **system()**

```
system(”rm *”);
```

Używaj znaku _ w nazwach zmiennych i funkcji

```
my $connection_status = 0;

sub read_many_lines { ... }
```

Stosuj czytelne konwencje

```
$ALL_CAPS_HERE      # stałe
$Some_Caps_Here     # globalne
$no_caps_here       # lokalne
function()          # funkcja (każda!)
```

Nie używaj znaku / jako ogranicznika wyrażeń regularnych, jeśli wzorzec zawiera znaki / lub \

```
if($filename =~ m(^/home/)) { ... }
```

Używaj operatorów **and** i **or** zamiast **&&** i **||**

```
open(FILE,”>new”) or die ”can’t create\n”;
```

Zawsze sprawdzaj wynik funkcji odwołujących się do systemu

```
open(FILE,”>new”) or die ”can’t create\n”;
```

Łam argumenty operatora **tr**

```
tr    (abc)
      (ABC);
```

Bądź miły dla innych..... :)

Bądź konsekwentny - każda konwencja jest lepsza, niż brak konwencji.

Poezja Perla (by Sharon Hopkins)

```
#! /usr/bin/perl


APPEAL:
listen(please,please);
open yourself,wide;
join(you,me);
connect(us,together);
tell me;
do something if distressed;
read(books,$poems,stories) until peaceful;
study if able;
sort your feelings,reset goals,
   seek(friends,family,anyone);
select (always), length(of_days);
```

Wskazówki co do wydajności

...nie zawsze w zgodzie z zaleceniami stylistycznymi

Wskazówki co do wydajności

Unikaj indeksowania - używaj foreach zamiast for, jeśli to tylko możliwe.

Wskazówki co do wydajności

Używaj haszy, jeśli musisz przeszukiwać tablicę - hash jest zawsze szybszy od przeszukiwania liniowego

Wskazówki co do wydajności

Unikaj printf, jeśli wystarczy print

Wskazówki co do wydajności

Nie używaj eval wewnątrz pętli - umieść pętlę wewnątrz eval.

Wskazówki co do wydajności

Stosuj next if na początku pętli, szczególnie przed chomp

Wskazówki co do wydajności

Unikaj wyrażeń regularnych z wieloma kwantyfikatorami - rozbijaj je na krótsze i rozsądnie posortowane wyrażenia

Wskazówki co do wydajności

W wyrażeniach regularnych maksymalizuj nieopcjonalne literały (brzmi dziwnie, prawda?)

Wskazówki co do wydajności

Unikaj wywołań funkcji w ciasnych pętlach

Pakiety
Moduły
Klasy

3 grzechy główne programistów

LENISTWO
PYCHA
NIECIERPLIWOŚĆ

Lenistwo...

kiedy używasz „kopiuj-wklej” zamiast napisać funkcję albo pętlę...

Pycha...

kiedy piszesz funkcję lub pętlę, zamiast użyć „kopiuj-wklej”...

Niecierpliwość...

kiedy piszesz własny kod, zamiast poszukać już gotowego...

Pakiet

- mechanizm pozwalający utworzyć nową przestrzeń nazw
- jeden plik źródłowy może zawierać wiele pakietów
- jeden pakiet może rozciągać się na wiele plików źródłowych
- najczęściej jednak:
  PAKIET == PLIK ŹRÓDŁOWY

Pakiet

- brak wskazania pakietu → pakiet main
- większość przeszukań przestrzeni nazw → czas kompilacji
- wyjątki: funkcja eval, interpolacja, itp.
- włączenie pakietu (przestrzeni nazw) → deklaracja package

Pakiet

- składnia:

```
package NAZWA;
```

- deklaruje, że blok, procedura lub plik źródłowy należy do przestrzeni nazw NAZWA

Pakiet

- konwencja: pakiet należy do przestrzeni nazw o nazwie takiej jak nazwa zawierającego go pliku
- oddziałuje na nazwy zmiennych globalnych i deklarowanych jako local
- nie oddziałuje na zmienne deklarowane jako my

Pakiet

- zwyczajowo umieszczany w pliku pakiet.pm
- odwołania do bytów innych pakietów → kwalifikacja:

```
pakiet::funkcja();
$pakiet::skalar;
```

- użycie pakietu w innym pakiecie (np. w pakiecie main):
  - deklaracja use
  - deklaracja require

Przykładowy pakiet

```
#! /usr/bin/perl

# pack.pm

package pack;

sub fun {
    print "fun z pack\n";
}

return 1;
```

Wynik inicjalizacji pakietu – bardzo ważne!

Tablica symboli pakietu

- przechowywana w haszu o nazwie %pakiet::
- tablica symboli pakietu main: %main:: lub %::
- klucze: identyfikatory znane w pakiecie
- wartości: dane typu typeglob
- dane typeglob: *symbol

Tablica symboli pakietu

- *symbol identyfikuje wszystkie byty o identyfikatorze symbol, a więc `$symbol, @symbol, %symbol`, ...

- dostęp do wartości tablicy symboli:

```
my *symbol = $main::{”identyfikator”}
```

Przegląd tablicy symboli

```
foreach $nazwasym (sort keys %main::) {
   my *sym = $main::{$nazwasym};
   print ”skalar \$$nazwasym\n” if defined $sym;
   print ”tablica \@$nazwasym\n” if defined @sym;
   print ”hasz \%$nazwasym\n” if defined %sym;
}
```

typeglob – utożsamienie nazw

```
*zmienna1 = *zmienna2
```

od tej chwili wszystko, do czego uzyskuje się dostęp poprzez zmienna1, osiągalne jest też przez symbol zmienna2 czyli:

$zmienna1 jest tożsame z $zmienna2;
@zmienna1 jest tożsame z @zmienna2;
%zmienna1 jest tożsame z %zmienna2;

typeglob – częściowe utożsamienie nazw

```
*zmienna1 = \$zmienna2
```

- operator \ wytwarza odwołanie do swojego argumentu
- utożsamienie $zmienna1 i $zmienna2, ale nie @zmienna1 i @zmienna2

Inicjowanie i kończenie pakietu

- blok BEGIN: wykonywany tak wcześnie, jak to możliwe (przed analizą reszty pliku pakietu)
- jeśli istnieje więcej, niż jeden blok BEGIN - są wykonywane wg kolejności definicji

Inicjowanie i kończenie pakietu

- blok END: wykonywany tak późno, jak to możliwe (gdy interpreter kończy działanie, ale nie w wyniku działania sygnału)
- jeśli istnieje więcej, niż jeden blok END - są wykonywane w kolejności odwrotnej, niż kolejność definicji
- stąd wynika, że pary BEGIN END są zagnieżdżone w oczekiwany (?) sposób

Inicjowanie i kończenie pakietu

```
#! /usr/bin/perl

die ” DWA ”;

END { print ” TRZY ”; }

BEGIN { print ” RAZ ”; }
```

Inicjowanie i kończenie pakietu

- bloki BEGIN będą wykonywana nawet wtedy, gdy tylko sprawdzasz składnię (opcja -c interpretera)
- blok END nie może zakładać, że jakakolwiek część skryptu się wykonała

Inicjowanie i kończenie pakietu

- zanim użyjesz niepewnej zmiennej, sprawdź, czy ma dobrą wartość...
- ...bo możesz narobić sobie kłopotów:

```
END { system ”rm -rf $dir”; }
```

Autoładowanie

- nie możesz wywołać niezdefiniowanej funkcji...
- ...chyba że masz w pakiecie funkcję o nazwie AUTOLOAD
- w takim przypadku wywołanie nieznanej procedury spowoduje:
  - podstawienie jej kwalifikowanej nazwy do zmiennej $AUTOLOAD
  - wywołanie procedury AUTOLOAD z tymi samymi parametrami, co „oryginał”
- możesz więc udawać istnienie nieznanych procedur

Autoładowanie

```
#! /usr/bin/perl

# emulowanie poleceń systemu

sub AUTOLAD {
  my $program = $AUTOLOAD;
  $program =~ s/.*::/; # utnij kwalifikację
  system($program, @_);
}

date();
ls('-l');
```

Dyrektywa **use subs**

```
use subs LISTA;
```

powoduje uznanie symboli wymienionych na liście za identyfikatory procedur, nawet jeśli nie są znane ich definicje, np.:

```
use subs qw(proc1 proc2 proc3);
proc(1,2,3);
```

Emulowanie poleceń systemu raz jeszcze

```
#! /usr/bin/perl

use subs qw(date ls);
sub AUTOLAD {
  my $program = $AUTOLOAD;
  $program =~ s/.*::/; # utnij kwalifikację
  system($program, @_);
}

date;
ls '-l';
```

Moduł

- moduł to pakiet nadający się do ponownego użycia
- zdefiniowany w pliku o tej samej nazwie, co pakiet
- konwencja: nazwa rozpoczyna się wielką literą
- moduł może eksportować wybrane symbole do przestrzeni nazw pakietu do używającego
- może funkcjonować jako definicja klasy

Eksportowanie symboli z modułu

```
#! /usr/bin/perl
# mod.pm

package Mod;
require Exporter;
@ISA = qw(Exporter); #dziedziczenie
@EXPORT = qw(proc1 proc2); #domyślne
@EXPORT_OK = qw($scalar @array);
#dozwolone
```

Importowanie symboli z modułu

```
! /usr/bin/perl

use Mod; # import domyślny
use Mod qw($scalar); # import na żądanie
use Mod (); bez importu
```

Funkcja **require**

```
require WYRAŻENIE;
```

- dołącza i wykonuje dokładnie raz kod odnaleziony w pliku o nazwie określonej przez wyrażenie;
- przeszukuje katalogi wymienione w `@INC`
- funkcja kończy się sukcesem, jeśli dołączany plik zwróci wartość „prawda”
- nie importuje jakichkolwiek symboli z dołączanego pliku
- powoduje wykonanie bloku `BEGIN`

require vs **use**

```
require Mod;
$result = Mod::fun();
```

```
use Mod qw(fun);
$result = fun();
```

Zastępowanie funkcji wbudowanych

- funkcja zaimportowana z modułu może zastąpić funkcję wbudowaną
- nie ma to nic wspólnego z przeciążaniem!
- możliwe do wykonania bez importu:

```
*open = \&myopen;
```

- dostęp do zastąpionej funkcji jest możliwy poprzez pseudopakiet CORE:

```
CORE::open(FILE,$file);
```

OBIEKTY i KLASY w PERLU

Repetytorium obiektowości

- obiekt – struktura danych z określonym zbiorem zachowań
- obiekt jest wcieleniem (instancją) klasy (instance of class)
- klasa – definiuje metody (instancyjne), które odnoszą się do wszystkich obiektów klasy (instance methods)
- klasa – definiuje metody niezależne od istnienia jej wcieleń – metody statyczne
- niektóre metody służą wyłącznie tworzeniu instancji – konstruktory

Repetytorium obiektowości

- metody mogą wykonywać operacje na wielu obiektach
- klasa może dziedziczyć metody z klasy macierzystej (bazowej, nadklasy, superklasy)
- wywołanie metody, której nie ma w pewnej klasie spowoduje jej wyszukanie w klasach macierzystych
- obiekt winien być traktowany jako czarna skrzynka
- uwaga: w Perlu nie ma środków jawnej specyfikacji interfejsu (bądź rozsądny i dobrze dokumentuj kod)

Obiekty w Perlu

- OBIEKT – byt (najczęściej hasz), do którego istnieje odwołanie i który wie, do jakiej klasy należy
- KLASA – pakiet, którego funkcje manipulują obiektami (a więc pełnią rolę metod)
- METODA – funkcja, która jako pierwszego argumentu spodziewa się odwołania do obiektu

Obiekt – byt z odwołaniem

- funkcja `bless` – „ochrzczenie” bytu jako należącego do klasy

```
bless BYT
bless BYT, PAKIET
```

- (jeśli brak drugiego argumentu, uznaje się, że `bless` dotyczy aktualnego pakietu)

Konstruktor

- Perl nie ma specjalnej składni wyróżniającej konstruktor
- konstruktor tworzy anonimowy byt, inicjuje go, chrzci i zwraca odwołanie do niego

Konstruktor

- typowy konstruktor:

```
package Class;
sub new {
   my $objref = {};
   bless $objref;
   return $objref;
}
# krócej
sub new { return bless {}; }
```

Wywołanie metody w konstruktorze

```
sub new {
   my $self = {};
   bless $self;
   $self->_init();
   return $self;
}
```

- podkreślenie sygnalizuje metodę „prywatną”, chociaż Perl nie zna takiego pojęcia
- wywoływana jest metoda init z pakietu aktualnego lub z pakietu klasy bazowej

Klasy

- wewnątrz metod odwołania do bytu odbywają się normalnie, jakby nie były ochrzczone
- poza pakietem należy traktować byt jako nieprzejrzysty
- Perl nie dostarcza składni definicji klas → klasa to pakiet z definicją metod
- Perl nie dostarcza składni definicji dziedziczenia → tablica ISA
- każdy element tablicy jest nazwą pakietu (czyli klasy), w którym będzie rekurencyjnie poszukiwana brakująca metoda (namiastka dziedziczenia)

Klasy

- pakiety (klasy) przeszukiwane są w takiej kolejności, w jakiej wymienione są w tablicy ISA
- namiastka wielodziedziczenia
- jeśli Perl znajdzie brakującą metodę, zapamięta jej położenie (przyśpieszenie powtórnego odwołania)
- jeśli jej nie znajdzie, ale znajdzie funkcję AUTOLOAD, to ona zostanie wywołana z w pełni kwalifikowaną nazwą metody
- jeśli nie ma funkcji AUTOLOAD, Perl będzie szukał metody w pakiecie (klasie) UNIVERSAL – metody „ostatniej szansy”
- jeśli wszystko zawiedzie, Perl zgłosi błąd

Klasy

- Perl zna tylko dziedziczenie metod – za dziedziczenie własności odpowiada sama klasa
- większość klas przechowuje swoje własności w anonimowym haszu
- dostęp do własności nie jest częścią Perla!
- hasz = przestrzeń nazw (?)
- własności nie są w jakikolwiek sposób deklarowane → po prostu albo istnieją w haszu, albo nie
- oznacza to, że własności mogą być tworzone *ad hoc*

Klasy

- Perl nie zna składni definicji metody – metoda to funkcja
- Perl zna składnię wywołania metody
- metody statyczne zakładają, że pierwszym argumentem jest nazwa pakietu (przekazywana jako łańcuch)
- Konstruktory → metody statyczne
- metody statyczne mogą być użyte do przeszukiwania wszystkich obiektów danej klasy
- metody instancyjne zakładają, że pierwszym argumentem jest odwołanie do obiektu
- zwyczajowo „przesuwa się go” (shift) do zmiennej $this albo $self

Klasy

- Perl nie ma składni odróżniania metod instancyjnych i statycznych
- oznacza to, że ta sama funkcja może pełnić obie role
- funkcja ref

```
ref WYRAŻENIE
```

- zwraca **true**, jeśli WYRAŻENIE jest odwołaniem, a napis pusty w wypadku przeciwnym; jeśli WYRAŻENIE odnosi się do „ochrzczonego” bytu, zwracana jest nazwa jego pakietu. Jeśli WYRAŻENIE odnosi się do bytu standardowego, zwracana jest nazwa typu: REF, SCALAR, ARRAY, HASH, CODE, GLOB.

Przykładowa klasa

```
package Cplx;
sub new{
    my $class = shift;
    my $self = {};
    if(defined $_[0]){
        $self->{Re} = shift;
    }
    if(defined $_[0]){
        $self->{Im} = shift;
    }
    bless($self,$class);
}
sub Re{
    my $self = shift;
    return $self->{Re};
}
sub Im{
    my $self = shift;
    $self->{Im};
}
1;
```

Wywołanie metod

- forma pierwsza – obiektu pośredniego

```
# METODA KLASA_LUB_INSTANCJA LISTA

$cplx = new Cplx (1.0, 1.0);
$re = Re $cplx;
```

Wywołanie metod

- forma druga – obiektowa

```
# KLASA_LUB_INSTANCJA->METODA(LISTA)

$cplx = Cplx->new(1.0, 1.0);
$re = $cplx->Re();
```

Uwaga – nawiasów nie wolno pominąć!

Niszczenie obiektu

- obiekt jest automatycznie niszczony w chwili usunięcia ostatniego odwołania do niego
- można przejąć kontrolę nad usuwaniem obiektu definiując w klasie metodę DESTROY