

Perl –wprowadzenie

Dr inż. Maciej Miłostan, Instytut Informatyki,
Politechnika Poznańska



Co to jest Perl

- Język programowania
- Prawdziwa perła wśród języków interpretowano-kompilowanych
- Sposobem użycia zbliżony do języków skryptowych – programy napisane w Perlu często nazywa się skryptami (zwłaszcza w kontekście automatyzacji prac administratorów systemów)
- Elastyczny i potężny w swej funkcjonalności – jak szwajcarski scyzoryk
- Język to Perl, perl to kompilator, nigdy PERL



Typowe użycie

- Przetwarzanie tekstów
- Serwisy internetowe – zwłaszcza oparte o CGI
- Automatyzacja zadań administracyjnych
- Współpraca z bazami danych
- Inne aplikacje internetowe



Główne przypadki użycia

- Human Genome Project*
- NASA* – język został stworzony przez Larryego Walla, gdy tam pracował

*Źródło: An Introduction to Perl Programming, Dave Cross, Magnum Solutions Ltd



Filozofia Perla*

- Na wszystko jest więcej niż jeden sposób
- Trzy cnoty programistów:
 - Lenistwo
 - Niecierpliwość
 - Pycha
- Dzielcie się i radujcie!

*Źródło: An Introduction to Perl Programming, Dave Cross, Magnum Solutions Ltd



Skąd pobrać?

- Jest dystrybuowany z większością dystrybucji linuksowych, jeśli używasz linuksa lub Mac OS-a, to prawdopodobnie już go masz
- Oficjalna strona Perla:
 - <http://www.perl.org>
- Wersje pod windows:
 - [ActiveState Perl](#)
 - [Strawberry Perl](#)
- Najnowsza wersja Perla to 5.16

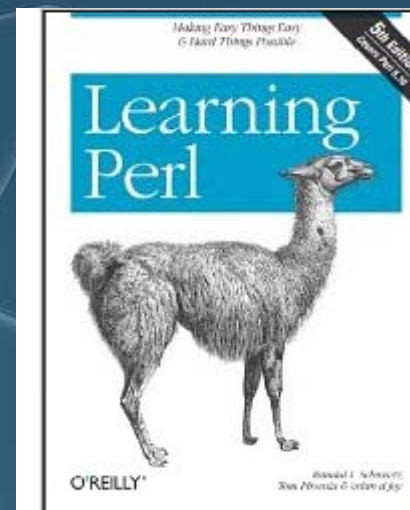
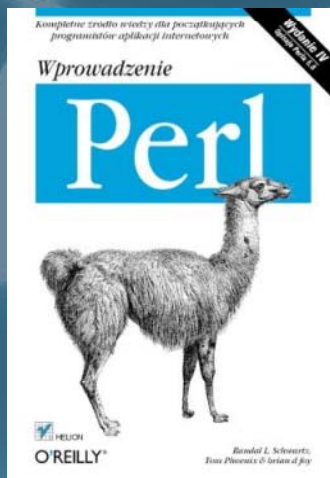
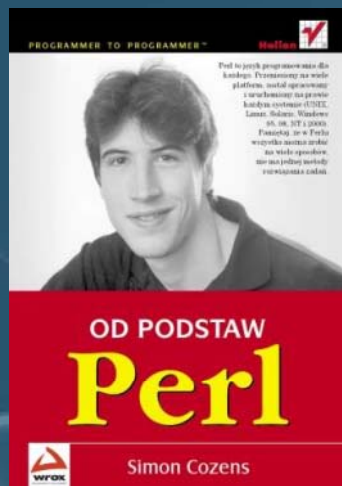


Materiały do nauki

- Książki elektroniczne, instruktaże, prezentacje:

<http://www.perl.org/learn.htm>

- Książki w języku polskim i angielskim:





- Niniejszy wykład został opracowany na podstawie:

An Introduction to Perl Programming,
Dave Cross, Magnum Solutions Ltd

A wide-angle photograph of a beach at sunset. The sky is filled with soft, golden light from the setting sun, which is partially obscured by clouds. The ocean waves are breaking gently onto the shore, creating white foam. In the foreground, a large, dark, gnarled piece of driftwood sits on the dark sand. The overall mood is serene and contemplative.

Pierwsze kroki



Pierwszy program

- Implementacja. W pliku tekstowym o nazwie hello.pl (lub innej dowolnej) wpisujemy:

```
print "Hello world\n";
```

- Uruchomienie. Z linii poleceń (w katalogu, gdzie zapisaliśmy plik programu) wydajemy polecenie:

```
perl hello.pl
```



Uruchamianie programu w Perlu

- Jeśli w systemie Linux/Unix chcemy uruchamiać program Perla odwołując się do nazwy (bez

Uwaga: Tutaj nie ma żadnych spacji ani innych białych znaków

nazwy (bez jawnego wywołania polecenia perl), to:

- W pierwszej linii

A tutaj jest koniec wiersza – znak nowej linii

```
#!/usr/bin/perl
```

- Plik programu musi mieć w systemie plików ustawioną flagę programu wykonywalnego:

```
chmod +x hello.pl
```

- Uruchomienie: `./hello.pl`



Komentarze w Perlu

- Komentuj swój kod – ułatwi to jego zrozumienie, gdy np. sięgniesz po niego za kilka lat lub komuś go udostępnisz
- Komentarz zaczyna się od hash-a (#)
- Komentarz kończy się na końcu linii



Komentowanie bloków kodu

- W trakcie „debug-owania” kodu użyteczne jest „wykomentarzowanie” całego fragmentu kodu (kilku linii)
- Perl nie ma w swojej składni takiej opcji, ale można wykorzystać mechanizmy preprocesora POD:

```
=for comment
```

```
print " Pierwsze miejsce\n";
```

```
print " Drugie miejsce\n";
```

```
=cut
```



Opcje wiersza poleceń

- Polecenie perl jest wyposażone w szereg opcji kontrolujących sposób wykonania
- Dla przykładu: -w włącza „warnings” (ostrzeżenia), używanie tej opcji w nowszych wersjach nie jest zalecane, zamiast tego lepiej do kodu dodać:

```
use warnings;
```

A photograph of a white laptop computer resting on a brown leather sofa. The laptop screen is on, displaying a 3D architectural rendering of a building. To the right of the laptop is a large, light-colored stuffed rabbit with long ears and a red body. The scene is dimly lit, with the laptop screen providing the primary light source. The text 'Źródła pomocy' is overlaid in white on the laptop screen.

Źródła pomocy



Pomoc w perlu

- Perl posiada rozbudowaną dokumentację
- Dokumentacja jest dostępna za pomocą polecenia `perldoc`:

```
perldoc perl
```

```
perldoc perltoc ← spis treści
```

- Dokumentacja on-line:

<http://perldoc.perl.org>



Perldoc-kilka użytecznych stron

- [perlintro](#)
- [perldata](#)
- [perlsyn](#)
- [perlfaq](#)
- [perlstyle](#)
- [perlcheat](#)

A scenic landscape featuring a calm lake in the foreground. Several trees with sparse green leaves are partially submerged in the water. In the background, there are dark, rugged mountains with patches of snow on their peaks. The sky is blue with scattered white clouds. The text "Zmienne i typy danych" is overlaid in the center of the image.

Zmienne i typy danych



Zmienne

- Co to jest zmienna?
 - Miejsce, w którym możemy składować nasze dane
- Zmienna (miejsce składowania danych) musi mieć swoją nazwę:
 - By można było w niej umieszczać dane
 - By można było z niej pobierać składowane w niej dane



Nazwy zmiennych

- W Perlu nazwa zmiennej może zawierać:
 - Znaki alfanumeryczne i znak podkreślenia
 - Nazwy zmiennych użytkownika nie mogą zaczynać się od liczb
 - Nazwy zaczynają się od specjalnego znaku „interpunkcyjnego” (\$, #, @) (ang. punctuation mark) oznaczającego typ danych



Typy danych

- Trzy główne typy danych: skalary, tablice, tablice asocjacyjne
- Zmienne poszczególnych typów danych zaczynają się od różnych symboli:
 - Skalary (ang. scalar) od \$
 - Tablice (ang. array) od @
 - ang. Hash (Tablice asocjacyjne) od %
- Więcej informacji o typach na dalszych slajdach



Deklaracje zmiennych

- W Perlu nie trzeba deklarować zmiennych, ale można i warto to robić
- Deklarowanie zmiennych zmniejsza ryzyko błędów związanych z:
 - Literówkami
 - Zakresem (ang. scope) – obszarem/obrębem ważności danej zmiennej (np. obszar pojedynczej funkcji, obszar całego programu, blok kodu itp.)
- Wymuszenie kontroli typów – użycie pragmy `strict`:

```
use strict;  
my $zmienna;
```



Zmienne skalarne

- Przechowują pojedynczy „egzemplarz” danych (a single item of data)
 - `my $imie = "Jan";`
 - `my $kim_jestem = 'Wielkim hakerem';`
 - `my $lat_pracy_nad_soba = 42;`
 - `my $liczba_mniejsza_od_1 = 0.001;`
 - `my $bardzo_duza_liczba = 3.27e17;`
#3.27 razy 10 do potęgi 17



Konwersja typów

- Perl dokonuje automatycznej konwersji pomiędzy łańcuchami znaków i liczbami zawsze kiedy jest to konieczne
- Dodawanie liczby całkowitej (ang. integer) i zmiennoprzecinkowej (ang. float):
 - `$suma= $lat_pracy_nad_soba + $liczba_mniejsza_od_1;`
- Umieszczanie liczby w łańcuchu znaków:

```
print "$imie mówi, 'Liczba lat
pracy nad sobą to $sum.'\n";
```




Łańcuchy znaków

- Pojedyncze i podwójne cudzysłowie:
 - W pojedynczych cudzysłowach łańcuchy nie są przetwarzane

```
my $price = '$9.95';
```

- W podwójnych są przetwarzane

```
my $incline = "24 widgets @ $price each\n";
```

Użyj znaku \ (backslash), żeby wypisać znak specjalny:

```
print "He said \"The price is  
\$300\"";
```



Lepsze cytowania

- Normalne cytowanie może wyglądać okropnie:

```
print "He said \"The price is  
\ $300\"";
```

- Ładniejszą alternatywą jest:

```
print qq(He said "The price  
is\ $300");
```

- To działa też dla pojedynczych cytowań

```
print q(He said "That's too  
expensive");
```



Wartości niezdefiniowane

- Zmienna skalarna bez przypisanej wartości przyjmuje wartość specjalną „undef”
- Do sprawdzenia, czy zmienna jest zdefiniowana możemy użyć funkcji:

`defined()`

- Przykład:

```
if (defined($my_var)) { ... }
```



Tablice, zmienne tablicowe

- Tablica zawiera uporządkowaną listę wartości skalarnych

```
my @fruit = ('apples', 'oranges',  
            'guavas', 'passionfruit',  
            'grapes');
```

```
my @magic_numbers = (23, 42, 69);
```

```
my @random_scalars = ('mumble',  
                      123.45,  
                      'dave cross',  
                      -300, $name);
```



Elementy tablicy

- Dostęp do indywidualnych elementów tablicy:

```
print $fruits[0];# prints "apples"
```

Uwaga: Indeksy zaczynają się od zera

```
print $random_scalars[2];  
# prints "dave cross"
```

- Zauważ użycie \$, co wynika z faktu, że indywidualny element tablicy jest wartością skalarną



Fragmenty tablicy

- Dostęp do wybranych elementów z tablicy – fragmentu tablicy (ang. array slice)

```
print @fruits[0,2,4];  
# prints "apples", "guavas",  
# "grapes"  
print @fruits[1..3];  
# prints "oranges", "guavas",  
# "passionfruit"
```

- Zauważ użycie @ - tym razem uzyskujemy dostęp do więcej niż jednego elementu



Przypisywanie wartości do elementów tablicy

```
$array[4] = 'something';  
$array[400] = 'something else';  
@array[4, 7 .. 9] = ('four', 'seven',  
                    'eight', 'nine');  
@array[1, 2] = @array[2, 1];  
@10 = (1 .. 10);  
@abc = (a .. z);  
($x, $y) = ($y, $x);
```



Ustalanie rozmiaru tablicy

- Wyrażenie:

`$#array` określa numer ostatniego elementu tablicy `@array`

- A zatem liczba elementów w tablicy jest równa:

```
 $count = $#array + 1
```

- To samo możemy uzyskać prościej:

```
 $count = @array;
```




Hash – tablica asocjacyjna

- Inicjalizacja za pomocą list:

```
%french = ('one', 'un',  
          'two', 'deux',  
          'three', 'trois');
```

- Łatwiejsza w zrozumieniu inicjalizacja za pomocą symbolu: => ("fat comma")

```
%german = (one    => 'ein',  
          two     => 'zwei',  
          three  => 'drei');
```



Dostęp do wartości hash-y

- Przykład

```
$three = $french{three};  
print $german{two};
```

- Tak jak w przypadku tablic używamy tutaj dolara (\$) by dostać się do skalara (pojedynczej wartości)



Fragmenty hash-y

- Analogicznie do tablic:

```
print @french{'one','two','three'};  
# wypisuje "un", "deux" & "trois"
```

- Zwróć uwagę, że w tym przypadku używamy @



Hash – przypisywanie wartości

- `$hash{foo} = 'something';`
- `$hash{bar} = 'something else';`

- `@hash{'foo', 'bar'} = ('something', 'else');`
- `@hash{'foo', 'bar'} = @hash{'bar', 'foo'};`



Hash - ograniczenia

- Hash-y nie można sortować
- Nie ma odpowiednika wyrażenia `$#array` dla hash-a
- Polecenie `print %hash` jest nieużyteczne
- Obejścia tych ograniczeń zobaczymy później



Zmienne specjalne

- Perl obfituje w zmienne specjalne
- Wiele z nich używa znaków interpunkcji jako nazw
- Część ma nazwy pisane WIELKIMI literami
- Zmienne specjalne są udokumentowane w perlvar (perldoc perlvar)



Zmienne domyślne (default)

- Wiele operacji w Perlu ustawia wartość zmiennej `$_` albo używa jej wartości jeśli inna nie została explicite podana np.

```
print; # wypisze wartosc $_
```
- Jeśli pewien fragment kodu w Perlu wygląda tak jakby zapomniano w nim o zmiennych, to prawdopodobnie używa `$_`
- Można o tym myśleć jak o wyrażeniach z języka naturalnego „to”, „tamto”, „tego”.



Użycie \$_

```
while (<FILE>)           Wczytanie linii z pliku do $_
{
    if (/regex/)        Wyszukanie wzorca w $_
    { print; }          Wypisanie wartości $_
}
```

- Trzy przypadki użycia



Tablica specjalna @ARGV

- Informacje o opcjach wiersza poleceń są przekazywane w tablicy specjalnej @ARGV

- Wywołanie programu z arumentami:

```
perl printargs.pl Ala Basia Henryk
```

- Kod, który wypisze liczbę argumetnów i ich wartości:

```
my $num = @ARGV;  
print "$num arguments: @ARGV\n";
```



Specjalny hash

- `%ENV` zawiera wartości zmiennych środowiskowych wyeksportowanych do programu
- Klucze są nazwami zmiennych np.

```
print $ENV{PATH};
```

Operacje wejścia/wyjścia





Operacje wejścia/wyjścia

- Programy zwykle są bardziej użyteczne, jeśli można coś do nich wczytać, lub można coś wypisać
- Trochę informacji o obsłudze plików itp. poznamy później, ale na początek warto pokazać podstawowe przykłady



Wyjście (Output)

- Najprościej użyć print albo say:
print "Hello world\n";
- Prawda, że proste



Wejście (Input)

- Najprościej wczytać dane do Perla ze standardowego wejścia:
- `$input=<STDIN>;`
- Wyrażenie `<...>` oznacza „wczytaj dane z uchwytu pliku (ang. filehandle)”
- `STDIN` oznacza uchwyt standardowego wejścia



Kompletny przykład

```
#!/usr/bin/perlprint  
'What is your name: '; $name = <STDIN>;  
print "Hello $name";
```

Operatory i funkcje





Operatory i funkcje

- Co to są operatory i funkcje?
- Takie „byty”, które wykonują „prace”
- Procedury wbudowane w Perla umożliwiające manipulowanie danymi
- Większość języków silnie rozróżnia operatory i funkcje w Perlu ten podział jest nieco rozmyty
- Zobacz perlop i perlfunc (w perldoc)



Operatory arytmetyczne

- Standardowe operatory arytmetyczne:
 - dodawanie(+), odejmowanie (-)
 - mnożenie(*), dzielenie(/)
- Trochę mniej standardowe:
 - operacja modulo(%) – reszta z dzielenia
 - Potęgowanie (**)

```
$predkosc = $odleglosc / $czas;
```

```
$objetosc = $dlugosc * $glebokosc * $wys;
```

```
$pole = $pi * ($r ** 2);
```

```
$nieparzyste = $liczba % 2;
```



Skrócone zapisu

- `$total = $total + $amount;`
`$total += $amount;`
- `$x++;` # znaczy `$x += 1` lub `$x = $x + 1`
- `$y--;` # znaczy `$y -= 1` or `$y = $y - 1`
- **Subtelna różnica pomiędzy `$x++` i `++$x`**



Operacje na łańcuchach

- **Konkatenacja (łączenie łańcuchów)**

```
$name = $firstname . ' ' . $surname;
```

- **Powtórzenie**

```
$line = '-' x 80;
```

```
$police = 'hello ' x 3;
```

- **Zapisy skrócone**

```
$page .= $line; # $page = $page . $line
```

```
$thing x= $i; # $thing = $thing x $i
```



Operatory testowania plików

- Umożliwiają sprawdzanie atrybutów plików:
- -e \$plik czy plik istnieje
- -r \$plik czy plik jest do odczytu
- -w \$plik czy mamy prawa do zapisu w pliku
- -d \$plik czy plik jest katalogiem
- -f \$plik czy plik jest normalnym plikiem
- -T \$plik czy plik jest plikiem tekstowym
- -B \$plik czy plik jest plikiem binarnym



Funkcje

- Mają dłuższe nazwy niż operatory
- Mogą posiadać więcej argumentów niż operatory
- Argumenty są podawane po nazwie funkcji
- W celu uzyskania informacji o funkcjach wbudowanych w Perla zobacz sekcję `perlfunc` (korzystając z `perldoc`) w dokumentacji Perla



Funkcje przyjmują wartości

- Funkcje mogą przyjmować („zwracać”) wartości skalarne, listowe lub puste

- `$age = 29.75;`

- `$years = int($age);`

- `@list = ('a', 'random',
'collection', 'of',
'words');`

- `@sorted = sort(@list);`

- `# a collection of random words`



Funkcje operujące na łańcuchach

- Długość łańcucha:

```
$len = length $a_string;
```

- Wielkie litery na małe – `lc` i na odwrót – `uc`

```
$string = 'MiXeD CaSe';
```

```
print "$string\n", uc $string,  
      "\n", lc $string;
```

- Zobacz także:

```
ucfirst i lcfirst
```




Więcej funkcji łańcuchowych

- chop usuwa i zwraca ostatni znak z łańcuch

```
$word = 'word';  
$letter = chop $word;
```
- chomp usuwa ostatni znak tylko, gdy jest to znak nowej linii i zwraca odpowiedni wartość prawda (true) lub fałsz (false)



Podłańcuchy

- substr zwraca podłańcuch z danego łańcucha:

```
$string = 'Hello world';  
print substr($string, 0, 5);  
# prints 'Hello'
```

- W Perlu do podłańcucha można przypisać wartość!!! (bardzo mało języków daje taką możliwość)

```
substr($string, 0, 5) =  
'Greetings'; print $string;  
# prints 'Greetings world'
```



Funkcje numeryczne

- abs – wartość bezwzględna
- cos, sin, tan – standardowe funkcje trygonometryczne
- exp – e podniesione do podanej potęgi
- log - logarytm naturalny (o podstawie e), *logarytm10_z_n=log(\$n)/log(10);*
- rand – przyjmuje wartość losową
- sqrt – pierwiastek kwadratowy



Operacje na tablicach

- push dodaje element na końcu tablicy (jakby tablica była stosem)
- pop zdejmuje ostatni element z tablicy (jak ze stosu)
- shift i unshift robią to samo, ale z początkiem tablicy
- Przykład:

```
push @array, $value;  
$value = pop @array;
```
-



Operacje na tablicach

- sort przekazuje posortowaną listę elementów (nie sortuje w miejscu)
- `@sorted = sort @array;`
- sort robi znacznie więcej, zobacz dokumentację (`perldoc -f sort`)
- reverse przekazuje listę odwróconą
- `@reverse = reverse @array;`



Tablice i łańcuchy

- **join** – łączy elementy tablicy w łańcuch, oddzielając poszczególne elementy zdefiniowanym ciągiem znaków

```
@array = (1 .. 5);  
$string = join ', ', @array;  
# $string is '1, 2, 3, 4, 5'
```

- **split** – tworzy tablicę na podstawie łańcucha znaków

```
$string = '1~2~3~4~5';  
@array = split(/~/, $string); # @array  
is (1, 2, 3, 4, 5)
```



Funkcje operujące hashami

- `delete` - usuwa parę klucz/wartość
- `exists` - „mówi” czy dany element istnieje w hash-u
- `keys` – podaje wartość wszystkich kluczy w hashu (w postaci listy)
- `values` – zwraca listę wszystkich wartości zapisanych w hash-u



Operacje plikowe

- **open** – otwiera plik i wiąże go z uchwyttem pliku

```
open(my $file, '<', 'in.dat');
```

- **Otwarty plik, można odczytać:**

```
$line = <$file>; # jedna linia
```

```
@lines = <$file>; # wszystkie linie
```

- **I zamknąć:**

```
close($file);
```




Inne operacje na plikach

- **read** – odczytuje podaną liczbę bajtów do bufora

```
$bytes = read(FILE, $buffer, 1024);
```

- **seek** – umożliwia skok do dowolnej pozycji w pliku

```
seek(FILE, 0, 0);
```

#drugi argument to pozycja w bajtach a trzeci, określa czy chodzi o pozycję względną

- **tell** – podaje bieżącą pozycję w pliku

- `$where = tell FILE;`

- **truncate** – obcina plik do podanego

rozmiaru: `truncate FILE, $where;`



Zapis do pliku

- **Otwórz plik w trybie do zapisu**

```
open my $file, '>', 'out.dat';# nadpisz
```

```
open my $file, '>>', 'out.dat';# dołącz
```

- **Zapisuj przy użyciu print:**

```
print $file "some data\n";
```

- **Zauważ brak przecinka**



Czas

- **Funkcje:**
 - `time` – przyjmuje wartość w sekundach od północy 1. stycznia 1970r.
`$now = time;`
 - `localtime` – konwertuje powyższe, do bardziej przyjaznej wartości
`($sec, $min, $hour, $mday, $mon, $year, $yday, $isdst) = localtime($now)`
 - `$mon` is 0 to 11
 - `$year` is years since 1900
 - `$yday` is 0 (Sun) to 6 (Sat)
 - `$isdst` prawda, gdy czas letni (Daylight Saving Time),



Czas - przykłady

```
@time_bits = localtime(time);
```

```
@time_bits = localtime;
```

```
($d, $m, $y) = (localtime)[3 .. 5];
```



Formatowanie daty i czasu

- Można operować na indywidualnych wartościach zwróconych przez `localtime`, albo
- Wygodniej:

```
print strftime('%Y-%m-%d', localtime);  
print strftime('%d %B %y', localtime);  
# z wykorzystaniem POSIX.pm
```
- Format jest zgodny ze standardem POSIX, tak jak komenda `date` w linuxsie/uniksie



Instrukcje warunkowe

- Umożliwiają kontrolowanie przebiegu wykonania kodu
- W zależności od weryfikowanych warunków program będzie przebiegał różne ścieżki wykonania
- Kontrolowaną jednostką wykonania jest blok instrukcji
- Bloki instrukcji obejmuje się nawiasami klamrowymi {...}

Instrukcje warunkowe





Instrukcje warunkowe

- Bloki warunkowe są kontrolowane poprzez ewaluację wartości wyrażenia, które może być prawdą lub fałszem
- Kiedy wyrażenie jest prawdą?



Co to jest prawda?

- W Perlu łatwiej wskazać co jest fałszem (false):
 - 0 (liczba zero)
 - "" (pusty łańcuch)
 - undef (wartość niezdefiniowana)
 - () (pusta lista)
 - Wszystko inne jest prawdą



Operatory porównania

- Porównanie wartości w jakiś sposób:
- Czy są równe
 $x == y$ lub $x eq y$
 $x != y$ lub $x ne y$
- Czy jeden jest większy od drugiego?
 $x > y$ lub $x gt y$
 $x >= y$ lub $x ge y$
- Także $<$ (lt) i $<=$ (le)



Przykłady porównań

- `62 > 42` # true
- `'0' == (3 * 2) - 6` # true
- `'apple' gt 'banana'` # false
- `'apple' == 'banana'` # true(!)
- `1 + 2 == '3 bears'` # true
- `1 + 3 == 'three'` # false
- `'apple' eq 'banana'` # false



Operatory boolowskie

- Łączenie dwóch lub więcej wyrażeń logicznych w jedno
- `EXPR_1 and EXPR_2`
- Prawda (`true`) jeśli zarówno `EXPR_1` i `EXPR_2` są prawdziwe
- `EXPR_1 or EXPR_2`
- Prawda (`true`) jeśli co najmniej jedno z wyrażeń `EXPR_1` lub `EXPR_2` są prawdziwe
- Alternatywna składnia `&&` dla `and` i `||` dla `or`
- W zależności od notacji inna klejność ewaluacji



Short-circuit

- `EXPR_1 or EXPR_2`
- Operator musi dokonać ewaluacji `EXPR_2` tylko, gdy `EXPR_1` jest fałszywe (`false`)
- Możemy ten fakt wykorzystać do uproszczenia kodu:

```
open FILE, 'something.dat' or die "Can't  
open file: $!";
```

```
@ARGV == 2 or print $usage_msg;
```



if

- if – (jeżeli) nasza pierwsza instrukcja warunkowa
- if (EXPR) { BLOCK }
- Wykonaj BLOCK tylko jeżeli EXPR jest prawdą

```
if ($name eq 'Doctor') {  
    regenerate();  
}
```



if ... else ...

- if ... else ... - rozszerzony if
- if (EXPR) { BLOCK1 } else { BLOCK2 }
- Jeśli EXPR jest true, wykonaj BLOCK1, w przeciwnym razie BLOCK2

```
if ($name eq 'Doctor') {  
    regenerate();  
} else {  
    die "Game over!\n";  
}
```



if ... elsif ... else ...

- ```
if ($name eq 'Doctor') {
 regenerate();
}
elsif ($tardis_location eq $here) {
 escape();
}
else { die "Game over!\n";
}
```



Peçle





# Pętla while

- Powtarzaj ten sam blok kodu dopóki warunek jest spełniony

```
while (EXPR) { BLOCK }
```

- ```
while ($num) {  
    print "Count down: $num \n";  
    $num--;  
}
```



Until

- Odwrotność while – wykonuj, aż warunek zostanie spełniony

```
until ($num == 12) {  
  print "Incrementing value\n";  
  dosomething();  
  $num++;  
}
```



for

- Pętla for
- Podobnie jak w C
- `for (INIT; EXPR; INCR) { BLOCK }`
- Wykonaj INIT, jeśli EXPR jest fałszem, to zakończ, w przeciwnym razie wykonaj BLOCK, potem wykonaj INCR i ponownie sprawdź EXPR



for

- Przykład:

```
for ($i = 1; $i <= 10; $i++) {  
    print "$i squared is ", $i * $i, "\n";  
}
```

- W Perlu tej pętli używa się zadziwiająco rzadko



foreach

- Dla każdego elementu
- Foreach umożliwia prostszą iterację po elementach list
- `foreach VAR (LIST) { BLOCK }`
- Dla każdego elementu z listy LIST, przypisz VAR wartość elementu i wykonaj BLOCK

```
foreach $i (1 .. 10) {  
    print "$i squared is ", $i * $i, "\n";  
}
```



forerach

- **Przykład:**

- ```
my %months = (Jan => 31,
 Feb => 28,
 Mar => 31,
 Apr => 30,
 May => 31,
 Jun => 30,
 ...);
```

```
foreach (keys %months) {
 print "$_ has $months{$_} days\n";
}
```



# Użycie pętli while

- Wczytywanie danych ze standardowego wejścia

```
while (<STDIN>) { print;}
```

- Powyższy zapis jest równoznaczny:

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





# Przerywanie pętli

- next –przeskocz do następnej iteracji pętli
- last – wyskocz z pętli
- redo – przeskocz, żeby wystartować tę samą iterację pętli jeszcze raz



Podprogramy



# Podprogramy - subroutines

- „Samo wystarczalne” mini-programy w obrębie Twojego programu
- Ułatwiają powtórzenie kodu
- Podprogramy to inaczej funkcje i procedury
- Podprogramy mają nazwę i zawierają blok kodu

```
sub NAME {
 BLOCK
}
```



# Przykład

```
sub exterminate {
 print "Ex-Ter-Min-Ate!!\n";
 $timelords--;
}
```



# Wywołanie podprogramu

- `&exterminate;`
- `exterminate();`
- `exterminate;`
- Ostatni przykład działa tylko, kiedy funkcja została w odpowiedni sposób predeklarowana (ang. *predeclared*) – perl musi się jakoś dowiedzieć, że jest to funkcja a nie zmienna



# Przekazywanie argumentów

- Funkcje są bardziej użyteczne, jeśli można przekazać do nich argumenty:  
`exterminate('The Doctor');`
- Argumenty trafiają do predefiniowanej tablicy `@_` dostępnej wewnątrz procedury
- ```
sub exterminate {  
    my ($name) = @_; print "Ex-Ter-Min-Ate  
    $name\n"; $timelords--;  
}
```



Wiele argumentów

- @_ jest tablicą, więc może zawierać wiele argumentów
- ```
sub exterminate {
 foreach (@_) {
 print "Ex-Ter-Min-Ate $_\n";
 $timelords--;
 }
}
```



# Wywołania podprogramów

- Subtelna różnica pomiędzy `&my_sub` i `my_sub()`
- `&my_sub` przekazuje zawartość `@_` do wywoływanego podprogramu

Dla zobrazowania:

```
sub first { &second };
sub second { print @_ };
first('some', 'random', 'data');
```

- Zwykle nie stosuje się takich konstrukcji jak ta na przykładzie





# Przekazanie przez referencję lub wartość

- Do podprogramów możemy przekazywać wartości argumentów albo
- „Same zmienne” (referencje do zmiennej) – zmiana wartości przekazanego argumentu powoduje zmianę zawartości zewnętrznej w stosunku do podprogramu zmiennej
- Perl pozwala wybrać sposób przekazania



# Przekazanie przez referencję lub wartość

- Simulacja przekazania wartości (by value): `my ($arg1, $arg2) = @_;`
- Zmiana `$arg1` i `$arg2` nie wpływa na wartości poza podprogramem
- Simulacja przekazania przez referencje
- `$_[0] = 'whatever';`
- Uaktualnienie `@_` powoduje aktualizację zewnętrznych wartości



# Przekazywanie wyników

- Przekazania wyniku możemy dokonać za pomocą `return` w przeciwnym razie, będzie to ostatnia ewaluowana wartość

```
sub exterminate {
 if (rand > .25) {
 print "Ex-Ter-Min-Ate $_[0]\n";
 $timelords--;
 return 1;
 } else {
 return;
 }
}
```



# Przekazywanie wyników

- ```
sub maximum {  
  if ($_[0] > $_[1]) {  
    $_[0];  
  } else {  
    $_[1];  
  }  
}
```



Przekazywanie list

```
sub exterminate {  
  my @exterminated;  
  foreach (@_) {  
    if (rand > .25) {  
      print "Ex-Ter-Min-Ate $_\n";  
      $timelords--;  
      push @exterminated, $_;  
    }  
  }  
  return @exterminated;  
}
```

A photograph of two dolphins swimming in clear, turquoise water. The dolphins are sleek and dark-colored, with their dorsal fins visible above the surface. The water is bright and clear, with some ripples and reflections. The text "Wyrażenia regularne" is overlaid in the center of the image.

Wyrażenia regularne



Wyrażenia regularne

- Wzorce pasujące do tekstów, łańcuchów znaków
- Coś na kształt symboli maski (wildcard-ów) znanych z wiersza poleceń
- „Mini-język” wewnątrz Perla
- Klucz do potęgi Perla w dziedzinie analizy tekstów
- Czasami nadużywane!
- Udokumentowane w **perldoc perlre**



Operator dopasowania (match)

- `m/PATTERN/MODIFIER` - operator dopasowania (match operator)
- Domyślnie przetwarza `$_`
- W kontekście skalarnym przekazuje wartość `true` jeśli udało się dopasować wzorzec/wyrażenie
- W kontekście listowym „zwraca” listę „dopasowanych” tekstów
- `m` jest opcjonalne jeśli używa się ograniczników /
- z `m` można używać dowolnych ograniczników
- `MODIFIER` określa sposób modyfikacji zachowania operatora: np. `i` – oznacza ignorowanie wielkości znaków



Przykład

```
while (<FILE>) {  
    print if /foo/;  
    print if /bar/i;  
    print if m|http://|;  
}
```



Zamiana/podstawienie /substytucja (substitution)

- `s/PATTERN/REPLACEMENT/` - operator substytucji
- Domyślnie operuje na `$_`
- W kontekście skalarnym przekazuje wartość `true` jeśli udało się dopasować wzorzec/wyrażenie
- W kontekście listowym podje liczbę zastąpień
- Można używać dowolnych ograniczników



Przykład

```
while (<FILE>) {  
    s/teh/the/gi;  
    s/freind/friend/gi;  
    s/sholud/should/gi;  
    print;  
}
```



Operator powiązania (binding operator)

- Jeśli chcemy by `m` lub `s` działało na czymś innym niż `$_`, to musimy użyć operatora powiązania (binding operator)

```
$name =~ s/Maciej/Miłosz;
```



Metaznaki

- Umożliwiają tworzenie wzorów dopasowujących nie tylko ciągi literalne
 - `^` - dopasowuje początek łańcucha
 - `$` - dopasowuje koniec łańcucha
 - `.` – dopasowanie dowolnego znaku (za wyjątkiem `\n`)
 - `\s` – pasuje do dowolnego znaku białego
 - `\S` – pasuje do dowolnego znaku nie będącego znakiem białym



Więcej metaznaków

- `\d` – pasuje do dowolnej cyfry
- `\D` – pasuje do dowolnego znaku nie będącego cyfrą
- `\w` – pasuje do każdego „słownego” znaku
- `\W` – pasuje do każdego "nie-słownego" znaku
- `\b` – pasuje do ograniczników słów np. `abc\b` pasuje do `abc!` i nie pasuje do `abcd`
- `\B` – pasuje do wszystkiego co nie jest ogranicznikiem słowa np. `perl\B` pasuje do *perlere* a nie do *perl to język*



Przykład

```
while (<FILE>) {  
    print if m|^http|;  
    print if /\bper\b/;  
    print if /\S/;  
    print if /\$\d\.\d\d/;  
}
```



Kwantyfikatory

- Specyfikują liczbę wystąpień
- ? - dopasuj zero lub jeden raz
- * - dopasuj zero lub więcej razy
- + - dopasuj jeden lub więcej razy
- {n} - dopasuj dokładnie n razy
- {n,} - dopasuj n lub więcej razy
- {n,m} – dopasuj między n i m razy



```
while (<FILE>) {  
    print if /whiske?y/i;  
    print if /so+n/;  
    print if /\d*\.\d+/  
    print if /\bA\w{3}\b/;  
}
```



Klasy znaków

- Definiowanie klasy znaków
Np. samogłosek w j. angielskim:
/[aeoiu]/
- Ciągły zakres znaków:
/[A-Z]/
- Dopasowanie elementów z dopełnienia zbioru
/[^A-Za-z]/ #Dopasowuje wszystko co nie jest literą



Alternatywa

- Użyj | żeby jeden ze zbiorów możliwych opcji
`/rose|martha|donna/i;`
- Do grupowania można użyć nawiasów
`/^(rose|martha|donna)$/i;`



Przechwytywanie dopasowań

Nawiasy wykorzystujemy również do przechwytywania dopasowań częściowych

Przechwycone dopasowania przechowywane są w zmiennych \$1, \$2, itd.

```
while (<FILE>) {  
    if (/^(\\w+)\\s+(\\w+)/) {  
        print „Pierwszym słowem było $1\\n”;  
        print „Drugim słowem było $2”;  
    }  
}
```



Przekazywanie dopasowanych elementów

- Dopasowane elementy są przekazywane również gdy operator match zostanie użyty w kontekście listowym

```
my @nums = $text =~ /(\d+)/g;  
print "Znalazłem następujące liczby całkowite:\n";  
print "@nums\n";
```



Więcej informacji

- perldoc perlre
- perldoc perlretut

SMART MATCHING





Smart Matching

- Mechanizm wprowadzony w Perl 5.10
- Operator dopasowania dający bardzo duże możliwości
- Zrób to co mam namyśli (Do What I Mean)
- Sprawdza operandy
- Podejmuje decyzje jakie dopasowanie zastosować



Operator Smart Match

- Nowy operator

`~~`

- Zbliżony do operatora dopasowania (ang. binding operator)

`=~`

- Może być używany w zastępstwie ww. (`=~`)
- Wyrażenie:

`$some_text =~ /some regex/`

Można zastąpić

`$some_text ~~ /some regex/`



Inteligentniejsze dopasowania (smarter matching)

- Jeśli jednym z operandów jest wyrażenie regularne, to `~~` wykonuje dopasowanie do wzorca

- Ale w sposób inteligentny np.

`%hash ~~ /regex/`

dokona dopasowań na kluczach hash-y

`@array ~~ /regex/`

dokona dopasowań na elementach tablicy



Więcej inteligentnych dopasowań

- Sprawdzenie czy tablice są identyczne
`@array1 == @array2`

- Sprawdzenie, czy tablica zawiera dany skalar

`$scalar == @array`

- Sprawdzenie, czy skalar jest kluczem w tablicy asocjacyjnej (hash-u)

- `$scalar == %hash`



Inteligentne dopasowania skalarów

- Jaki rodzaj dopasowania wykona poniższe wyrażenie?
`$scalar1 ~~ $scalar2`
- To zależy
- Jeśli obydwa skalary wyglądają na liczby to `~~` zachowuje się jak `==`
- W przeciwnym razie `~~` zachowuje się jak `eq`

Korzystanie z modułów





Moduły

- Moduł to reużywalna „porcja” kodu
- Perl jest wyposażony standardowo w ponad 100 modułów (zobacz perldoc perlmodlib, a uzyskasz pełną listę)
- Perl posiada repozytorium ogólnodostępnych darmowych modułów - the Comprehensive Perl Archive Network (CPAN)
- <http://www.cpan.org>
- <http://search.cpan.org>



Wyszukiwanie modułów

- <http://search.cpan.org>
- Wyszukiwanie po:
 - Nazwie modułu
 - Nazwie dystrybucji
 - Autorze
- Uwaga: CPAN zawiera również nowsze wersje modułów standardowych



Instalowanie modułów (trudniejsza droga)

- Pobierz plik z dystrybucją
MyModule-X.XX.tar.gz
- Zdekompresuj - unzip
- \$ gunzip MyModule-X.XX.tar.gz
- Rozpakuj - untar
- \$ tar xvf MyModule-X.XX.tar
- Przejdź do rozpakowanego katalogu
- \$ cd MyModule-X.XX



Instalowanie modułów (trudniejsza droga)

- Wygeneruj Makefile
- `$ perl Makefile.PL`
- Skompiluj/zbuduj moduł
- `$ make`
- Przetestuj wynik kompilacji
- `$ make test`
- Instaluj moduł
- `$ make install`



Instalowanie modułów (trudniejsza droga)

- Uwaga: możesz potrzebować przywilejów administratora - root-a (root permissions), aby wykonać `make install`
- Użytkownik może mieć również osobistą bibliotekę modułów
`perl Makefile.PL PREFIX=~/.perl`
Konieczne jest dostosowanie `@INC`



Instalowanie modułów (łatwiejsza droga)

- Uwaga: Może nie działać poprzez firewall
- CPANPLUS.pm jest dystrybuowany z nowszymi Perl-ami
- Automatycznie przeprowadza proces instalacji
- Rozwiązuje zależności między modułami
- Może nadal wymagać uprawnień root-a



Instalowanie modułów (łatwiejsza droga)

- `cpanp`
[... jakiś tekst ...]
CPAN Terminal> `install Some::Module`
[... Jeszcze trochę tekstu ...]
CPAN Terminal> `quit`
- `Lub`
- `cpanp -i Some::Module`



Używanie modułów

- Dwa typy modułów:
 - funkcyjne i obiektowe
- Moduły funkcyjne eksportują nowe podprogramy/funkcje/procedury oraz zmienne do naszego programu
- Moduły obiektowe zwykle tego nie robią
- Rozróżnienie pomiędzy tymi rodzajami modułów nie jest sztywne (weźmy na przykład CGI.pm)



Używanie modułów funkcyjnych

- Domyślny import:
`use My::Module;`
- Import opcjonalny komponentów :
`use My::Module qw(my_sub @my_arr);`
- Import zdefiniowanych zbiorów komponentów:
`use My:Module qw(:advanced);`
- Użycie zaimportowanych komponentów:
`$data = my_sub(@my_arr);`



Używanie modułów obiektowych

- Użyj modułu:
- `use My::Object;`
- Utwórz obiekt:
- `$obj = My::Object->new;`
 - Uwaga: wywołanie `new` jest tylko konwencją
- Użyj metod obiektu:
`$obj->set_name($name);`



Użyteczne moduły standardowe

- constant
- Time::Local
- Text::ParseWords
- Getopt::Std
- Cwd
- File::Basename
- File::Copy
- POSIX
- CGI
- Carp
- Benchmark
- Data::Dumper



Użyteczne moduły niestandardowe

- Template
- DBI
- DBIx::Class
- DateTime
- HTML::Parser
- HTML::Tidy
- LWP
- WWW::Mechanize
- Email::Simple
- XML::LibXML
- XML::Feed
- Moose



Rodzaje zmiennych i zakresy ich użycia



Rodzaje zmiennych

- Zmienne leksykalne (ang. lexical variables)
my \$zmienna
Związana z blokiem kodu
- Zmienne „pakietowe” (ang. package variables)
our \$zmienna
Związana z pakietem/modułem



Zmienne leksykalne

- Zasięg widoczności:
 - Blok kodu ograniczony nawiasami (klamrowymi)
 - Plik źródłowy
- Leksykalna, bo zasięg (ang. scope) jest ograniczony tylko przez tekst



Pakiety

- Kod Perla składa się z pakietów
- Nowe pakiety tworzy się poprzez *package MyPackage;*
- Najlepiej o pakiecie myśleć jak o „przestrzeni nazw” (ang. namespace)
- Pakietów używa się, żeby zapobiegać konfliktom nazw z bibliotekami
- Domyślny pakiet to „main”



Zmienne pakietowe

- Egzystują w tablicy symboli pakietu
- Można się do nich odwoływać używając w pełni kwalifikowanych nazw:
 - \$main::zmienna
 - @pakiet::tablica
- Nazwa pakietu nie jest wymagana w obrębie pakietu
- Zmienna jest widoczna zewsząd w obrębie pakietu (lub zewsząd jeśli w pełni kwalifikowana)



Deklarowanie zmiennych pakietowych

- Zmienna może być predeklarowana za pomocą *our*

*our (\$doktor, @liczby,
%miejsce_urodzenia);*

- Lub (w starszych Perlach) przy użyciu *use vars*

*use vars qw(\$doktor @liczby
%miejsce_urodzenia)*



Zmienne leksykalne czy pakietowe

- Jakich zmiennych używać leksykalnych czy pakietowych?
- Prosta odpowiedź:
 - Zawsze używaj zmiennych leksykalnych
- Bardziej złożona odpowiedź:
 - Zawsze używaj zmiennych leksykalnych
 - Za wyjątkiem niewielkiej liczby wyjątków
- Jest jeszcze operator *local*
<http://perl.plover.com/local.html>



local

- *local* nie tworzy zmiennych lokalnych
- Cóż nazwa nie jest zbyt trafiona
- Co robi?
 - Tworzy lokalną kopię zmiennej pakietowej
- Może być użyteczna
 - W niewielkiej liczbie przypadków



Przykład użycia local

- `$/` jest zmienną pakietową
- Definiuje separator rekordów
- Możesz niekiedy chcieć go zmienić
- W takim przypadku ogranicz zakres zmian:

```
{ local $/ = "\n\n";  
    while ( <FILE> ) { ... }  
}
```



use strict / use warnings



Bezpieczne kodowanie

- Perl jest bardzo luźnym językiem programowania co niesie spore ryzyko
- Ryzyko możemy ograniczyć stosując dyrektywy specjalne:
use strict; / use warnings;
- Dobrze wyrobić sobie zwyczaj używania tych dyrektyw
- Żaden poważny programista Perla, nie koduje bez nich



use strict

- Kontroluje trzy rzeczy:
 - use strict `refs` – odwołania symboliczne
 - use strict `subs` – nazwy procedur / „gołe” słowa (ang. barewords)
 - use strict `vars` – deklaracje zmiennych
- use strict – włącza elementy kontroli dla wszystkich trzech ww.
- w niektórych przypadkach możemy chcieć wyłączyć sprawdzanie użyjemy wówczas
No strict



Use strict 'refs'

- W Perlu możliwe jest tworzenie zmiennych w następujący sposób:
\$what = 'dalek';
\$\$what = 'Karn';
zmienna \$dalek przyjmuje wartość 'Karn'
- Co jeśli 'dalek' został wprowadzony przez użytkownika?
- Ludzie często myślą, że to bardzo fajna cecha...
- ...co znaczy, że są w błędzie
- W powyższym przypadku lepiej użyć hash-y:
\$what = 'dalek'; \$alien{\$what} = 'Karn'



Use strict `subs`

- Nie możemy być „gołosłowni” (ang. bareword)
- Gołe słowo (bareword) to słowo, które nie ma innej interpretacji
- Np. słowo bez \$,@,%,&
- Słowo takie jest traktowane jako wywołanie funkcji albo cytowany łańcuch (quoted string)
- \$maciej=Milostan;
- Takie użycie może kolidować w przyszłości z zarezerwowanymi słowami – np. co jeśli później zdefiniujemy funkcję Milostan();



Use strict 'vars'

- Wymusza deklaracje zmiennych
- Zapobiega literówką
- Zawsze pamiętajmy o zakresie zmiennych



use warnings

- Ostrzeżenia przed podejrzanymi zwyczajami programistycznymi
- Przykłady typowych ostrzeżeń
 - Zmienna użyta tylko raz
 - Używanie zmiennych niezdefiniowanych/niezadeklarowanych
 - Zapis do uchwytów pliku otwartych w trybie tylko do odczytu
 - I wiele, wiele więcej



Lokalne wyłączenie ostrzeżeń

- Czasami nazbyt pracochłonne jest wyczyszczenie kodu, żeby uczynić go 100% czystym
- Możemy wyłączyć ostrzeżenia lokalnie
- Możemy wyłączyć specyficzne ostrzeżenia

```
{ no warnings 'deprecated';  
#wiekowy kod  
}
```
- Zobacz: `perldoc perllexwarn`



Referencje i złożone typy danych



Referencje

- Referencja w Perlu jest bytem zbliżonym do wskaźników w C i Pascalu (ale lepszym)
- Referencja jest unikalnym sposobem odwołania się do zmiennej
- Referencja zawsze pasuje do kontekstu skalarnego
- Referencja wygląda np. tak:
`SCALAR(0x20026731)`



Tworzenie referencji

- Umieść `\` przed nazwą zmiennej
 - `$scalar_ref = \ $scalar;`
 - `$array_ref = \@array;`
 - `$hash_ref = \%hash;`
- Teraz możesz referencje traktować jak każdy inny skalar:
 - `$var = $scalar_ref;`
 - `$refs[0] = $array_ref;`
 - `$another_ref = $refs[0];`



Tworzenie referencji

- [LIST] tworzy tzw. anonimową tablicę i przekazuje referencję do niej
 - `$aref = ['this', 'is', 'a', 'list'];$aref2 = [@array];`
- { LIST } tworzy anonimowy hash i przekazuje referencję do niego
 - `$href = { 1 => 'one', 2 => 'two' };`
 - `$href = { %hash };`



Tworzenie referencji

- Przykładowy kod:

```
@arr = (1, 2, 3, 4);$aref1 =  
\@arr;$aref2 = [ @arr ];print  
"$aref1\n$aref2\n";
```

- Wynik:

```
ARRAY(0x20026800)
```

```
ARRAY(0x2002bc00)
```

- Druga metoda **tworzy kopię tablicy**



Korzystanie z referencji do tablic

- Użyj `{$aref}` aby odzyskać tablicę, do której posiadasz referencję

- Cała tablica:

```
@array = @{$aref};
```

```
@rev = reverse @{$aref};
```

- Pojedynczy element

```
$elem = ${$aref}[0];
```

```
${$aref}[0] = 'foo';
```




Korzystanie z referencji do hash-y

- Użyj `{$href}` aby odzyskać hash do którego posiadasz referencję

- Cały hash

```
%hash = %{$href};
```

```
@keys = keys %{$href};
```

- Pojedynczy element

```
$elem = ${$href}{key};
```

```
${$href}{key} = 'foo';
```



Używanie referencji

- Użyj (->) aby uzyskać dostęp do elementu tablicy lub tablicy asocjacyjnej (hash-a)
- Zamiast $\{\$aref\}[0]$ możesz użyć $\$aref->[0]$
- Zamiast $\{\$href\}\{key\}$ możesz użyć $\$href->\{key\}$



Używanie referencji

- Aby dowiedzieć się do czego odwołuje się referencja możesz użyć ref:

```
$aref = [ 1, 2, 3 ];
```

```
print ref $aref; # Wypisze ARRAY
```

```
$href = { 1 => 'one', 2 =>  
'two' };
```

```
print ref $href; # Wypisze HASH
```



Po co używać referencji

- Przekazywanie parametrów
- Złożone typy danych



Przekazywanie parametrów

- Co to robi?

```
@arr1 = (1, 2, 3);  
@arr2 = (4, 5, 6);  
check_size(@arr1, @arr2);  
sub check_size {  
    my (@a1, @a2) = @_;  
    print @a1 == @a2 ? 'Yes' : 'No';  
}
```



Dlaczego to nie działa

- `my (@a1, @a2) = @_;`
- Tablice są łączone w `@_`
- Wszystkie elementy kończą w `@a1`
- Jak to możemy naprawić?
- Przekażmy referencje do tablic



Kolejna próba

```
@arr1 = (1, 2, 3);  
@arr2 = (4, 5, 6);  
check_size(\@arr1, \@arr2);  
sub check_size {  
    my ($a1, $a2) = @_;  
    print @$a1 == @$a2 ? 'Tak' :  
    'Nie';}
```



Złożone typy danych

- Kolejne dobre pole do użycia referencji
- Spróbujmy utworzyć tablicę 2D

```
@arr_2d = ((1, 2, 3),  
          (4, 5, 6),  
          (7, 8, 9));
```

@arr_2d zawiera (1, 2, 3, 4, 5, 6, 7, 8, 9)

- Jest to tak zwane spłaszczenie tablicy



Złożone typy danych

- Tablice 2D przy użyciu referencji
- `@arr_2d = ([1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]);`
- Ale jak uzyskać dostęp do indywidualnych elementów
`$arr_2d[1]` jest ref. do tab. (4, 5, 6)
- `$arr_2d[1]->[1]` jest elementem 5



Złożone typy danych

- Tablice 2D raz jeszcze
- `@arr_2d = [[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]];`
- `$arr_2d->[1]` jest ref. do tab. (4, 5, 6)
- `$arr_2d->[1]->[1]` jest elementem 5
- Można pominąć pośrednie strzałki
- `$arr_2d->[1][1]`



Więcej struktur danych

- Wyobraź sobie następujący plik danych
- Milostan, Maciej, Informatyk
Komorowski, Bronisław, Prezydent RP
Kowalski, Jan, Kowal
- Jaka struktura będzie odpowiednia do składowania takich danych
- Hash dla każdego rekordu
- Tablica rekordów
- Tablica hashy



Więcej struktur danych

- Tworzenie tablicy hash-y:

```
my @records;
```

```
my @cols = ('nazwisko', 'imie', 'praca');
```

```
while (<FILE>) {
```

```
    chomp;
```

```
    my %rec;
```

```
    @rec{@cols} = split /,/;
```

```
    push @records, \%rec;
```

```
}
```



Używanie tablicy hash-y

- ```
foreach (@records) {
 print "$_->{imie} ",
 "$_->{nazwisko} ".
 "is a $_->{praca}\n";}
```



# Złożone typy danych

- **Wiele możliwości**
  - Hash hash-y
  - Hash list
  - Wiele poziomów (lista hash-y hash-y, itd.)
- Zobacz przykłady w perldoc perldsc (the data structures cookbook)

# Sortowanie





# Sortowanie

- Perl ma funkcję `sort`, która jako argument bierze listę i ją sortuje
- `@posortowana = sort @tablica;`
- Zwróć uwagę, że sortowanie nie odbywa się w miejscu:  
`@array = sort @array;`





# Kolejność sortowania

- Domyślna kolejność jest określona wg. ASCII

```
@chars = sort 'e', 'b', 'a', 'd', 'c';
```

```
@chars ma ('a', 'b', 'c', 'd', 'e')
```

- Czasem daje to dziwne wyniki

```
@chars = sort 'E', 'b', 'a', 'D', 'c';
```

```
@chars ma postać ('D', 'E', 'a', 'b', 'c')
```

```
@nums = sort 1 .. 10;
```

```
@nums ma postać (1, 10, 2, 3, 4,
```

```
5, 6, 7, 8, 9)
```



## Bloki sortujące (ang. sorting block)

- Możemy zdefiniować własne kryteria sortowania używając tzw. „sorting blocks”
- `@nums = sort { $a <=> $b } 1 .. 10;`
- Perl umieszcza dwie wartości z listy w `$a` i `$b`
- Blok porównuje wartości i przyjmuje wartość `-1`, `0` lub `1`
- Operator `<=>` robi to dla liczb (`cmp` dla łańcuchów znaków)



# Proste przykłady sortowania

- `sort { $b cmp $a } @words`
- `sort { lc $a cmp lc $b } @words`
- `sort { substr($a, 4) cmp  
 substr($b, 4) } @lines`



# Funkcje sortujące

- Zamiast bloku sortującego możemy użyć funkcji

- `@words = sort dictionary @words;`

```
sub dictionary {
 # Don't change $a and $b
 my ($A, $B) = ($a, $b);
 $A =~ s/\W+//g;
 $B =~ s/\W+//g;
 $A cmp $B;
}
```



# Sortowanie wg. nazwisk i imion

```
my @names = ('Rose Tyler',
 'Martha Jones',
 'Donna Noble',
 'Anna Kowalska');

@names = sort sort_names @names;

sub sort_names {
 my @a = split /\s/, $a;
 my @b = split /\s/, $b;
 return $a[1] cmp $b[1] or $a[0] cmp $b[0]; }
}
```



```
sub sort_names {
 my @a = split /\s/, $a;
 my @b = split /\s/, $b;
 return $a[1] cmp $b[1] or $a[0] cmp
 $b[0]; }
}
```

- Powyższy kod będzie nieefektywny dla dużej liczby danych
- Wiele wywołań split na tych samych danych



# Efektywniejsz sortowanie

- Podziel (split) każdy rekord tylko raz  
`@split = map { [ split ] } @names;`
- Posortuj nową tablicę  
`@sort = sort { $a->[1] cmp $b->[1]  
or $a->[0] cmp $b->[0] } @split;`
- Połącz dane razem  
`@names = map { join ' ', @$_ }  
@sort;`



## Złożmy to razem

- Poprzedni kod możemy zapisać tak:  

```
@names = map { join ' ', @$_ } sort {
 $a->[1] cmp $b->[1] || $a->[0] cmp
 $b->[0] } map { [split] } @names;
```
- Wszystkie funkcje działają na wynikach poprzednich funkcji w łańcuchu





# Trasnformacja Schwartza

- Zamiast:

```
@data_out = sort { func($a) cmp
func($b) } @data_in;
```

- Efektywniej jest:

```
@data_out = map { $_->[1] } sort {
$a->[0] cmp $b->[0] } map {
[func($_), $_] } @data_in;
```

- Stary trik z Lispa
- Randal Schwartz

# Tworzenie modułów





# Po co pisać moduły

- Reużycie kodu
- Żeby „nie wywierać już otwartych drzwi”
- Łatwiejsze współdzielenie kodu między projektami
- Lepsze projektowanie, bardziej ogólne



# Podstawowy moduł

```
use strict;
use warnings;
package MyModule;
use Exporter;
our @ISA = ('Exporter');
our @EXPORT = ('my_sub');
sub my_sub { print "This is my_sub\n";}
1;
```



# Użycie MyModule.pm

```
use MyModule;
my_sub jest dostępna
do użycia w Twoim programie

my_sub();
Wypisze "This is my_sub()"
```



- Znaczna część kodu MyModule.pm dotyczy eksportu nazw procedur
- Pełna nazwa procedury
  - MyModule::my\_sub()
- Eksportowany skrót
  - my\_sub()



- Każda procedura „żyje” w pakiecie
- Domyślnym pakietem jest main
- Nowe pakiety wprowadza się przy użyciu słowa kluczowego package
- Pełna nazwa procedury to:
  - pakiet::nazwa\_procedury
- Nazwa pakietu może zostać pominięta w obrębie tego samego pakietu
- Jak w rodzinie 😊



# Używanie eksporterów

- Moduł `Exporter.pm` obsługuje eksport nazw procedur i zmiennych
- `Exporter.pm` definiuje procedure o nazwie `import`
- `Import` jest wywoływana automatycznie zawsze, gdy moduł jest używany
- `Import` umieszcza referencje do naszych procedur w tabeli symboli programu wywołującego





# Jak działa Exporter

- Jak MyModule wykorzystuje procedurę import Exporter-a?
- Korzystamy z mechanizmu dziedziczenia
- Dziedziczenie jest definiowane przy użyciu tablicy @ISA array
- Jeśli wywołujemy procedurę, która nie jest zdefiniowana w naszym module, wówczas również moduły zdefiniowane w @ISA są również sprawdzane
- A co za tym idzie Exporter::import jest wywoływana



# Symbole eksportu

- Skąd import wie, którą procedurę wyeksportować?
- Eksportowane elementy są definiowane w @EXPORT lub @EXPORT\_OK
- Automatyczne eksporty są zdefiniowane w @EXPORT
- Opcjonalne eksporty są definiowane w @EXPORT\_OK



# Eksportowanie zbioru symboli

- Zbiory eksportów można zdefiniować w `%EXPORT_TAGS`
- Kluczem jest nazwa zbioru
- Wartością jest referencja do tablicy nazw
- `our %EXPORT_TAGS = (advanced => [ qw( my_sub my_other_sub ) ] );`
- `use MyModule qw(:advanced);`  
`my_sub();my_other_sub();`



# Dlaczego używać @EXPORT\_OK?

- Umożliwia to twojemu użytkownikowi możliwość wyboru, które procedury zaimportować
- Mniejszy szansa na konflikty nazw
- Użycie @EXPORT\_OK jest bardziej preferowane niż @EXPORT
- Dokumentuj eksportowalne nazwy i zbiory



# Eksportowanie zmiennych

- Możesz również eksportować zmienne
- `@EXPORT_OK = qw($scalar,  
@array,  
%hash);`
- Mogą być one częścią zbioru eksportowego
- Wszelkie eksportowane zmienne muszą być zmiennymi pakietowymi (package variables)



# Pisanie modułów, łatwa droga

- Znaczne część kodów modułów jest zbliżona
- Nie pisz kodów, które są już napisane
- Kopiuj z istniejących modułów
- Zobacz też: `perldoc Module::Starter`

# Obiektowość





# Programowanie zorientowane obiektowo

- W programowaniu proceduralnym mamy procedury operujące na danych
- W programowaniu obiektowym klasy zawierają metody, które definiują ich akcję (operują na ich danych)
- Obiekty są instancjami klas
- Perl zapewnia wsparcie obiektowości na „wyśrubowanym” poziomie
- Łączy to co najlepsze z obydwu światów





# Perl zorientowany obiektowo

- Obiekt to moduł, który zachowuje pewne reguły
- Trzy reguły obiektó Perla:
  - Klasa jest pakietem
  - Obiekt jest referencją (zwykle do hash-a)
  - Metoda jest podprogramem/procedurą/funkcją (subroutine)
- *bless* „mówi” referencji jakiego rodzaju jest obiektem



# Prosty obiekt

```
package MyObject;
sub new {
 my $class = shift;
 my $name = shift;
 my $self = { name =>
 $name };
 return bless $self,$class;
}
```

```
sub get_name {
 my $self = shift;
 return $self->{name};
}
```

```
sub set_name {
 my $self = shift;
 $self->{name} = shift;
}
1;
```



## Użycie MyObject.pm

```
use MyObject;
my $obj = MyObject->new('Dave');
print $obj->get_name;
prints 'Dave'$obj->set_name('David');
print $obj->get_name;# prints 'David'
```



## Dodatowe informacje

- perldoc perlboot
- perldoc perltoot
- perldoc perlobj
- perldoc perlbot
- perldoc Moose (if it is installed)
- *Object Oriented Perl (Conway)*

# Komunikacja z bazami danych





# Bazy danych

- Grupa aplikacji wymagających komunikacji z relacyjnymi lub obiektowymi bazami danych rośnie w siłę
- Perl posiada narzędzia czyniące te odwołania tak prostymi jak to tylko możliwe
- Moduł DBI daje podstawy do komunikacji z większością współczesnych baz danych
- Warto korzystać z modułów DBI, lub modułów opartych na DBI



# Jak działa DBI

- Program używa DBI.pm
- Tworzy połączenie do bazy danych danego typu
- Moduł DBD zostaje załadowany
- DBD „tłumaczy” wywołania z API DBI na wywołania specyficzne dla danej bazy danych
- DBD konwertuje „zwrócone wartości do struktur Perlowych



## Łączenie się z bazą

```
use DBI;
```

```
my $dbh = DBI->connect(
 "dbi:mysql:$some_stuff", $user, $pass);
```

- "mysql" jest nazwą DBD  
DBD::mysql
- Teoretycznie można dokonać podmiany bazy na inną
- Wystarczy zmienić wiersz zawierający specyfikację połączenia






# Zapytania do bazy – przygotowanie zapytania

- Przygotowanie zapytania SQL-owego
  - `my $sth = $dbh->prepare( 'select name, genre from artist');`
  - `my $sth = $dbh->prepare( "select title, from song where artist = '$id'");`
- Sprawdź „zwracane wartości” ( na wypadek błędów składni (syntax errors))



# Zapytania do bazy – wykonanie zapytania

- Wykonaj zapytanie do bazy
- \$sth->execute
- Nadal sprawdzaj czy nie wystąpiły błędy



# Zapytania do bazy – pobranie wyników

- Pobierz dane otrzymane w wyniku zapytania:

```
while (my @row = $sth->fetchrow_array){
 print "@row\n";
}
```

- Pola w ramach krotki danych są przekazywane w takiej samej kolejności w jakiej zostały zdefiniowane w zapytaniu



# Inne funkcje pobierające wyniki

- `fetchrow_arrayref`
- `fetchrow_hashref` (kluczami są nazwy kolumn)
- `fetchall_arrayref`
- `fetch` (alias dla `fetchrow_arrayref`)
- Wiele nowych wciąż powstaje



# Uwagi ogólne

- Jeśli używasz metody fetch zwracającej tablicę, to nigdy nie używaj:  
"select \*"  
Z oczywistych powodów
- Jeśli używasz metody fetch zwracającej hash, to:
- Upewnij się, że wszystkie kolumny mają (unikalne) nazwy



# Insert, Update & Delete

- Zapytania/polecenia nie zwracające wyniku można wykonywać w ten sam sposób, co select:

```
my $sql = "update table1 set col1 = '$val'
 where id_col = $id";
```

```
my $sth = $dbh->prepare($sql);
```

```
$sth->execute;
```

- Ale można krócej:

```
$rows = $dbh->do($sql);
```



## Wiele insert-ów

```
while (<FILE>) {
 chomp;
 my @data = split;
 my $sql = "insert into tab values ($data[0],
 $data[1], $data[2])";
 $dbh->do($sql);}
```

- Rekompiluje SQL-a za każdym razem
- Bardzo nieefektywny sposób zapisu



# Dowiązanie danych

- Przygotuj zapytanie raz i używaj wiele razy (tzw. prepared statement)

```
my $sql = "insert into tab values (?, ?, ?)";
```

```
my $sth = $dbh->prepare($sql);
```

```
while (<FILE>) {
```

```
 my @data = split;
```

```
 bind_param(1, $data[0]);
```

```
 bind_param(2, $data[1]);
```

```
 bind_param(3, $data[2]);
```

```
 $sth->execute;}
```

- Dowiązanie (ang. binding) zapewnia odpowiednie cytowanie (ang. quotation) zmiennych





# Dowiązanie danych

- Można jeszcze prościej:
  - Przekaż dodatkowe parametry do execute

```
my $sql = "insert into tab
 values (?, ?, ?)";
my $sth = $dbh->prepare($sql);
while (<FILE>) {
 chomp;
 my @data = split;
 $sth->execute(@data);
}
```



# Nienazwane znaczniki miejsca (unnamed placeholders)

- Nienazwane odnośniki w zapytaniu, mogą być problematyczne i mylące:
- ```
my $sql = 'insert into big_table values( ?,  
?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  
?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)';
```
- Mamy dużą szansę na pobranie zmiennych w złej kolejności
- Takie zapytania utrudniają utrzymanie aplikacji i wprowadzanie ew. zmian (maintainability error)



Dowiązuj poprzez nazwy

```
my $sql = 'insert into big_table
(id, code, name, addr, email,
url, ... )
values (:id, :code, :name,
:addr, :email, :url,
... );
```

```
my $sth = $sql->prepare($sql);
```

```
$sth->bind_param(':id', $id);
```

```
$sth->bind_param(':code', $code);
```

```
# etc
```

```
$sth->execute;
```



Jeszcze prostsze dowiązania

- Składuj swoje dane w hashu

```
my %data = (id => 42,  
            code => 'H2G2',  
            ... );
```

i później...

```
foreach my $col (keys %data) {  
    $sth->bind_param(":$col",  
                    $data{$col});  
}
```



Dobre rady

- Czyń swoje życie tak prostym jak to możliwe
- Nie wpisuj na stałe w kodzie parametrów połączeniowych – lepiej stwórz plik konfiguracyjny, lub użyj zmiennej środowiskowej
- Składuj zapytania w zewnętrznym miejscu (np. plik) i odwołuj się do nich poprzez nazwę
- Używaj nazwanych dowiązań i „prepared statements” jeśli tylko możesz (zmniejszysz ryzyko błędów i ataków „sql injection”)



Przykładowy kod

```
my $dbh;  
sub run_sql {  
    my ($sql_statement, %args) = @_;  
    my $sql = get_sql($sql_statement);  
    $dbh = get_dbh() unless $dbh;  
  
    my $sth = $dbh->prepare($sql);  
    foreach my $col (keys %args) {  
        $sth->bind_param(":$col",  
                        $args{$col});  
    }  
    return $sth->execute;  
}
```



Mapowanie baz na obiekty

- ORM (ang. Object Relational Mapping)
 - Baza relacyjna a obiekty
 - Nazwy tabel (relacje) na klasy
 - Krotki na obiekty
 - Kolumny na atrybuty
 - Nie ma potrzeby samodzielnego pisania kodu SQL
- ORM w CPAN: Tangram, Alzabo, Class::DBI, DBIx::Class (lider)



Dodatkowe informacje

- `perldoc DBI`
- `perldoc DBD::*`
- `DBD::mysql`
- `DBD::Oracle`
- Etc...
- `perldoc DBIx::Class`

A landscape photograph of rolling green hills under a blue sky with scattered white clouds. Numerous sheep are grazing across the hillsides. A dirt road or path winds through the center of the scene. The text "BioPerl" is overlaid in the center of the image in a white, sans-serif font.

BioPerl



BioPerl

- Informacje o tym jak zainstalować moduły BioPerla, jak również przykłady użycia można znaleźć na stronie:

<http://www.bioperl.org>

- Na początek proponuję:

<http://www.bioperl.org/wiki/HOWTO:Beginners>



Projekt Bioperl

- Projekt BioPerl jest międzynarodowym zrzeszeniem twórców narzędzi open source w Perlu do celów analiz bioinformatycznych, genomicznych i z zakresu nauk o życiu
- Zainicjowany w 1995 przez grupę naukowców znużonych przepisami parserów do wyników BLASTA i różnych formatów sekwencji
- Wersja 0.7 została udostępniona w 2000 roku
- Bioperl 1.0 został stworzony w 2002
- Artykuł poświęcony bioperlowi opublikowano w listopadzie 2002: Satjich et al., 2002. The bioperl toolkit: perl modules for the life sciences. *Genome Research* 12: 1611-1618.)
- Najnowsza edycja 1.6.9 została udostępniona w kwietniu 2011



Bioperl moduły

- Core package (bioperl-live)
 - Pakiet podstawowy wymagany przez pozostałe
- Run package (bioperl-run)
 - Dostarcza metod do uruchamiania ponad 60 programów bioinformatycznych
- DB package (bioperl-db)
 - Podprojekt mający na celu umożliwienie składowanie danych sekwencyjnych i anotacji w relacyjnej bazie BioSQL
- Network package (bioperl-network)
 - Parsowanie i analiza danych interakcji białko-białko
- Dev package (bioperl-dev)
 - Nowo opracowywane metody itp..



Bioperl-zorientowany obiektowo

- Bioperl wykorzystuje możliwości dawane przez podejście obiektowe do tworzenia spójnego i dobrze udokumentowanego modelu obiektowego umożliwiającego interakcje z danymi biologicznymi.
- Przestrzeń nazw (ang. name space) w Bioperl
Pakiety Bioperl-a instalują wszystko w przestrzeni **Bio::** nazwaprzestrzeni.



Obiekty Bioperl

- **Obiekty obsługujące sekwencje**

- Obiekt sekwencja (Sequence)
- Obiekt Alignment
- Obiekt Location

- Inne obiekty:

struktury 3D, drzewa, drzewa filogenetyczne, mapy obiektów, dane bibliograficzne i obiekty graficzne



Operacje na sekwencjach

- Typowe operacje:
 - Dostęp do sekwencji
 - Formatowanie sekwencji
 - Alignment sekwencji i porównywanie sekwencji
 - Wyszukiwanie podobnych sekwencji
 - Porównywanie parami (Pairwise comparisons)
 - Alignment wielu sekwencji (Multiple Alignment)



Sekwencje jako obiekty

- Obiekty sekwencyjne: Seq, RichSeq, SeqWithQuality, PrimarySeq, LocatableSeq, LiveSeq, LargeSeq, SeqI
- Seq jest w BioPerlu głównym obiektem operującym na sekwencjach, może być używany do opisu sekwencji DNA, RNA lub białek
- Większość typowych operacji manipulujących sekwencjami może być wykoana za pomocą Seq.



Anotacje Sekwencji

- *Bio::SeqFeature* sekwencja może mieć wieke cech z nią związanych (ang. features) np. mogą to być takie obiekty jak gen (Gene), egzon (Exon), promotor (Promoter)
- *Bio::Annotation* Obiekt Seq może posiadać powiązany obiekt Annotation (używany do składowania komentarzy, linków do baz danych, odnośników literaturowych itp.)



Sequence Input/Output

- System *Bio::SeqIO* został zaprojektowany tak, aby umożliwić pobieranie, składowanie i przetwarzanie sekwencji z wielu źródeł w wielu różnych formatach

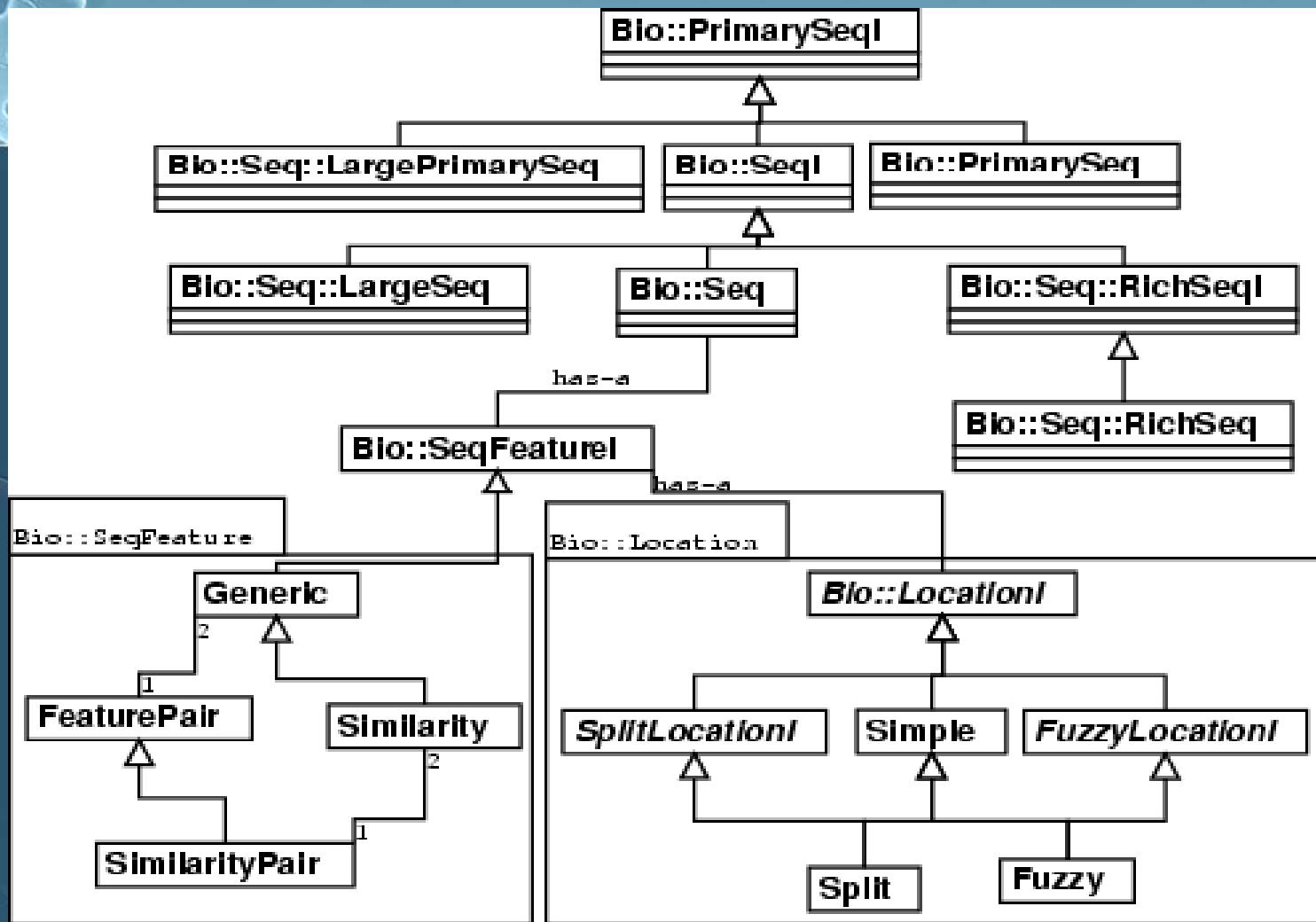


Diagram obiektów i interfejsów do analizy sekwencji



Dostęp do baz sekwencji

- Bioperl wspiera dostęp zarówno do baz zdalnych jak i lokalnych.
- Bioperl aktualnie potrafi obsłużyć pobieranie sekwencji z genbanku, genpept, RefSeq, swissprot-a, i baz EMBL



Formatowanie sekwencji

- SeqIO może odczytać strumień sekwencji w jednym formacie: Fasta, EMBL, GenBank, Swissprot, PIR, GCG, SCF, phd/phred, Ace, lub surowa sekwencja (raw/plain sequence), a następnie zapisywać je do innego pliku w innym formacie

```
use Bio::SeqIO;
$in = Bio::SeqIO->new('-file' => "inputfilename",
                    '-format' => 'Fasta');
$out = Bio::SeqIO->new('-file' => ">outputfilename",
                    '-format' => 'EMBL');
while ( my $seq = $in->next_seq() )
{ $out->write_seq($seq); }
```



Manipulowanie danymi sekwencji

- `$seqobj->display_id()`; # the human read-able id of the sequence
`$seqobj->subseq(5,10)`; # part of the sequence as a string
`$seqobj->desc()` # a description of the sequence
`$seqobj->trunc(5,10)` # truncation from 5 to 10 as new object
`$seqobj->revcom` # reverse complements sequence
`$seqobj->translate` # translation of the sequence
...



Alignment

- Wyszukiwanie „podobnych” sekwencji, Bioperl może wykonywać BLAST-a lokalnie lub zdalnie, a następnie parsować wyniki.
- Alignment dwóch sekwencji przy użyciu algorytmu with Smitha-Watermana (SW) lub blast-a
 - Algorytm SW został zaimplementowany w C i dołączony do BioPerla przy pomocy rozszerzenia XS.
- Multiple sequences alignment(Clustalw.pm, TCoffee.pm)
 - bioperl oferuje perlowy interfejs standardowych programów -clustalw i tcoffee.
- Bioperl udostępnia również parser wyników programu HMMER



Alignment - obiekty

- Wczesne wersje udostępniały UnivAIn, SimpleAlign
- Aktualna wersja używa SimpleAlign i AlignIO umożliwiając :
 - konwersje między formatami
 - ekstrakcję wyspecyfikowanego regionu w alignmencie
 - Generowanie sekwencji konsensusowych ...



- **Obiekty obsługujące sekwencje**
 - Obiekt sekwencja (Sequence)
 - Obiekt Alignment
 - Obiekt Location



Location

- `Bio::Locations`: kolekcja dość skomplikowanych obiektów
- Obiekt zaprojektowany do powiązania z obiektem `Bio::SeqFeature` w celu wskazania, gdzie w dużej sekwencji zlokalizowany jakiś charakterystyczny element (np. chromosom lub contig)



Operacje na strukturach 3D

- Obiekty StructureI and Structure::IO
http://www.bioperl.org/wiki/HOWTO:Bioperl_Objects#StructureI_and_Structure::IO
- Manipulacje plikami PDB



Podsumowanie

Co omówiliśmy:

- Typy danych proste i złożone
- Instrukcje sterujące
- Wyrażenia regularne
- Sortowanie
- Tworzenie własnych modułów i funkcji
- Dostęp do relacyjnych baz danych
- Podstawowe operacje na strukturach i sekwencjach przy użyciu BioPerla



Dziękuję za uwagę