RUBY

wg kursu Udemy

1. Obiekty

Ruby jest przede wszystkim językiem obiektowym (ale też wieloparadygmatowym).

Obiekt jest fundamentalną rzeczą w Rubim. Wszystko w Rubim jest obiektem (w innych językach tak nie jest).

Obiekt to swoista struktura, która przechowuje dane i zawiera metody, czyli funkcje do wykonania na tych danych lub w kontekście tych danych. Metody tworzą obiekty, operują na danych, pytają o stan danych itp. Np. przystanek jest obiektem. Ludzie na przystanku są obiektami. Autobus jest obiektem. Spóźnienie autobusu jest obiektem. W innych językach są obiekty i typy prymitywne. Np. liczba 2 w Rubim jest obiektem, czyli przechowuje dane, oraz czynności, które z tymi danymi możemy zrobić. Liczbę 2 można spytać, czy jest parzysta, czy nie, można z niej wyciągnąć następnik, można ją skonwertować na tekst itp.

1. Zmienne

Zmienna to wskazanie na obiekt. Zmienna nie jest obiektem. To, co się robi na zmiennych, robi się na obiekcie, który ona wskazuje.

Tworzenie zmiennych:

num = 1,201371

txt = „idź stąd!”

Zmienne można zmieniać.

Rubi to język dynamicznie typowany, co oznacza, że nie pilnuje typów wartości przypisywanych do zmiennej.

Nazwy zmiennych:

- nie mogą być za krótkie

- powinny jakoś informować o przeznaczeniu lub zawartości zmiennych

- nie powinny wprowadzać w błąd

mój\_kochany\_żuczek – tak wygląda zmienna składająca się z paru słów

nazwy powinny być po angielsku

Widoczność i typy zmiennych:

1. Zmienna lokalna: zmienna
2. Zmienna blokowa: zmienna
3. Zmienna globalna (?): $zmienna – jest widoczna w całym programie (?)
4. Zmienna klasy (?): @@zmienna – zmienna widoczna w całej klasie (?)
5. Zmienna instancji (?): @zmienna – zmienna widoczna w pojedynczym obiekcie

Przykłady w irb:

b = 3

b + 2

5

b.class -> fixnum

txt = „elo”

„elo”

txt2 = txt

txt2 >> “elo”

STRING = ZMIENNA ZAWIERAJĄCA TEKST

Txt.upcase! >> “ELO”

Txt > „ELO”

Txt2 >> „ELO” (tak jak txt!)

Obie zmienne wskazują na ten sam obiekt. Modyfikacja obiektu implikuje zmianę wartości OBU zmiennych. TO WAŻNA CECHA RUBIEGO!

Txt = „bum”

Txt >> „bum”

Txt2 >> „elo” !!!

1. Liczby

Na liczbach można wykonać wiele operacji. Działania takie jak mnożenie, dodawanie, dzielenie itd. nazywa się „operatorami”.

2 \*\* 7 to 2 do potęgi 7

\*\* znak potęgi

Nawiasy działają normalnie.

(2 + 2) \* 3 = 4 \*3 = 12

x = 8

x >> 8

x +=1 >> 9

x += 1to inaczej x=x+1

x-=1 to inaczej x=x-1

8

Liczby ujemne tworzy się przez postawienie przed liczbą znaku „-„ czyli np. – 64

Operacje na liczbach:

.abs – zwraca wartość bezwzględną, np.

-12.abs >> 12

.abs to metoda na obiekcie – czyli na np. -12

Inne metody:

.next – zwraca następnik, np. 8.next>> 9

.pred – zwraca poprzednik

.odd? - sprawdza, czy liczba jest nieparzysta

.even? – sprawdza, czy liczba jest parzysta

Liczby zmiennoprzecinkowe:

To liczby z ułamkiem dziesiętnym.

Np. 1.1

Typ to float.

1.1.class >> float

UWAGA!

8/3>> 2!!!!, a nie 2,66….

Ruby robi tak, bo 8 i 3 to liczby całkowite, więc myśli, że interesują nas tylko takie liczby.

Trzeba podzielić tak:

8.0/3

Albo: 8/3.0

Albo 8.0/3.0

RUBY DOBIERA SPÓJNY TYP WYNIKU

Metody na liczbach zmiennoprzecinkowych:

.floor – zwraca wartość całkowitą, ale zaniżając!, np. 3.8.floor>>3

.ceil – zwraca wartość całkowitą zawyżając!, np. 8.09.ceil>> 9

.round – zaokrągla liczbę

.round(1) – zaokrągla do podanego w nawiasie miejsca po przecinku

.to\_i – konwersja do liczby całkowitej (to integer)

.to\_f – konwersja do liczby zmiennoprzecinkowej (to float)

.to\_s – konwersja na tekst (to string)

Można konwertować tekst (string) na liczby, np.:

„7”.to\_f>>7.0

„7.0”.to\_i>>7

Float(„8.09”)>> 8.09 – inny rodzaj zapisu konwersji z tekstu na liczbę

Integer(12)>12

LICZBY ZMIENNOPRZECINKOWE NIE SĄ PRECYZYJNE! IM WIĘCEJ OPERACJI, TYM SĄ MNIEJ PRECYZYJNE!!!

Np. BANKI NIE OPERUJĄ NA LICZBACH ZMIENNOPRZECINKOWYCH (komentarz: dziwne, a jak zapisują np. 100 zł i 87 groszy, skoro nie 100,87?)

Klasa bigdecimal:

To klasa do precyzyjnej obsługi dużych liczb zmiennoprzecinkowych.

require ’bigdecimal’

x=BigDecimal.new(„1.0”) (liczbę podajemy jako tekst, bigdecimal skonwertuje to na własny format).

BigDecimal używa się tam, gdzie potrzebna jest bardzo duża precyzja w obliczeniach!

1. Tablice

Tablica to uporządkowana lista obiektów dowolnego typu.

[1, 2, 3] – tablica

[1, 9.0001, „strzała”] – tablica

[„okno”, 1.1, [„drzwi”]]

Tablice są obiektami, co oznacza, że posiadają metody.

.first>> wyświetla pierwszy element tablicy

.last – wyświetla ostatni element

.length>> wyświetla ilość obiektów w tablicy

.reverse – zwraca obiekty w odwrotnej kolejności. Ta metoda zwraca nowy obiekt, nie modyfikuje starego!

.reverse! - ta metoda modyfikuje tablicę, zmieniając kolejność elementów

To częsta konwencja w Rubim! Metoda bez wykrzyknika zwraca nowy obiekt, z wykrzyknikiem zmienia obiekt.

Każdy obiekt w tablicy ma swój indeks, czyli kolejność, którą zajmuje w tablicy.

Aby dostać się do danego indeksu robimy tak:

a[1]>> drugi obiekt.

Obiekty są indeksowane od 0!

a[-1] indeks z zerem przed nim zwraca obiekty od końca!

a[0..3] – pobiera wycinki tablicy – w tym wypadku od pierwszego do 4.

Manipulowanie zawartością tablicy:

Podmiana obiektu w tablicy

a = [1,2,3]

a[1]>>2

a[1] = 8

a[1]>> 8

nil to obiekt modelujący (oznaczający) wartość pustą

.delete\_at(2) – usuwa 3 argument

a << „d” – dodaje argument „d” na koniec tablicy

.unshift(„z”) – dodaje element na początek tablicy

b = a.shift – usuwa pierwszy element z tablicy a i go zwraca przypisując do zmiennej „b”

.delete(„nazwa obiektu”) – usuwanie obiektu z tablicy bez znajomości indeksu. Metoda ta usuwa wszystkie obiekty o tej nazwie.

.uniq – metoda usuwania duplikatów z tablicy. Metoda ta tworzy nową tablicę.

.uniq! – metoda modyfikująca istniejącą tablicę. Usuwa zduplikowane elementy.

.clear – usuwanie wszystkich elementów tablicy. Inną metodą zerowania jest przypisanie do zmienne oznaczającej tablicę tablicy pustej, np.

a = [1,2,3]

a = []

1. Metody z blokami

Przykłady wywołania metody z blokiem:

3.times do

Puts „hello”

End>>

hello

hello

hello

albo:

5.times {puts „elo”} – to, co pomiędzy nawiasami jest blokiem

elo

elo

elo

elo

elo

[1,2,3]. each do |a|

puts "Elo po raz #{a}"

end

Dla każdego elementu tablicy wykonywany jest blok:

Elo po raz 1

Elo po raz 2

Elo po raz 3

Alternatywna możliwość:

[1,2,3].each { |i| puts "Elo po raz #{i}"}

Najpopularniejsze metody używane na tablicach:

.each - służy do iteracji. Wyświetla wszystkie elementy tablicy

["a", "b", "c"].each do |um|

puts um

end

a

b

c

alternatywa:

["a", "b", "c"].each {|element| puts "element"}

.reverse\_each – wyświetla w kolejności odwrotnej

.reverese.each – najpierw odwraca, potem wyświetla. Efekt ten sam.

.sort – sortowanie tablicy

.map – jedna z najpopularniejszych metod, zwraca nową tablicę zmieniając treści z poprzedniej

[1,2,3].map { |n| puts n \* 2} – zapis w atomie

2

4

6

[1,2,3].map {|e| e \* 8} – zapis w irb

.all? – pyta, czy każdy element tablicy spełnia jakiś warunek

np. [1,2,3].all? {|n| puts n<10} – zapis w atomie

true

[1,2,3].all? {|n| n<10} – zapis w irb

.any? – analogicznie, przy czym metoda zwraca true albo false jeśli którykolwiek element spełnia warunek

.select – metoda wybiera wszystkie elementy tablicy spełniające warunek i tworzy nową tablicę z elementami spełniającymi warunek

[1,2,3,4,5].select {|e| e.even?} – nie wiem jak zapisać w atomie

[2,4]

.find - zwróci pierwszy obiekt spełniający warunek podany w bloku

.partition – tworzy dwie tablice. Jedną z obiektami dla których blok zwraca prawdę, a drugą z elementami, dla których blok zwraca fałsz

[1,2,3,4,5].partition {|e| e.even?}

[2,4], [1,3,5]

1. Stringi

String to łańcuch znaków. To typ danych, który umożliwia przechowywanie i operowanie na tekście. String to ciąg znaków w podwójnym lub pojedynczym cudzysłowie.

Podwójny cudzysłów reaguje na tzw. znaki specjalne. Przykładem znaku specjalnego jest „#”.

Aby znak specjalny potraktować w sposób dosłowny (bez wykonania go) należy przed nim postawić „\”, np. „\#{2+2}”; „Wal\”się”

\n – znak nowej linii, np. puts „Dwa\ndwa”

\t - tabulator

W Rubim stringi można łączyć (dodawać): „Elo” +” „ + „ziom”.

Można też je mnożyć, np. „osiem” \*3

Dostawanie się do znaku w stringu:

a = „Pejsy”

a[0] >>P

Zmiana znaków

a[0] = „ó”

albo na wiele znaków: a[1]=”uuuuuuu”

Pobieranie wycinka stringa:

a[0..1]

Skoro stringi są obiektami, to posiadają metody.

„elo”.upcase>> „ELO”

.downcase – na odwrót

.reverse – odwrócenie stringa

.length – liczy, ile znaków jest w stringu

Można łańcuchowo łączyć metody, np. : „elo”.upcase.reverse.

1. Symbole

Mają specyficzne zastosowania.

:test – symbol

Każdy obiekt w Rubim ma swój identyfikator. Stringi mają zawsze inny identyfikator. Nawet te same. Konkretny symbol ma zawsze ten sam identyfikator.

Sprawdzanie identyfikatora:

„bum”.object\_id

:bum.object\_id

Oznacza to, że symbol jest tworzony tylko raz i za każdym razem, gdy używamy symbolu, korzystamy z tego samego obiektu.

String jest zawsze nowym obiektem i jest przechowywany w innym miejscu pamięci.

Symboli używamy, gdy chcemy opisać stan za pomocą tekstu. Np. status zamówienia powinien być zapisany za pomocą symbolu – bo się nie zmieni. :shipped. Symboli używa się też do tworzenia indeksów w haszach.

1. Hasze

Hasze są też nazywane słownikami lub tablicami asocjacyjnymi. Hasz jest kolekcją danych – jak tablica – ale w haszu indeksem może być obiekt dowolnego typu. Hasz to sekwencja danych typu: klucz-wartość.

osoba = {„imię” => „Zenon”, „wiek” => „88”}

osoba[„imię”]>> „Zenon”

modyfikacja zawartości hasha

osoba[„imię”] = „Mariola”

Jako kluczy lepiej używać symboli, a nie stringów, ale użycie stringu nie jest błędem.

osoba = {:imię=>”Wacek”}

osoba[:imię]

Jeśli chcemy hasza indeksować za pomocą symboli, można go inaczej zapisać:

osoba = {imię: „Zenio”, wiek: 99}

Klucze w haszu mogą być dowolnego typu i mogą być mieszane:

miks = {1=> „jeden”, „dwa”=> :dwa, :trzy=>[1,23,0]}

.key – pytanie do hasza pod jakim kluczem znajduje się dana wartość:

miks.key(„jeden”)

.size- liczy ile obiektów znajduje się w haszu

.to\_a – konwertuje hasza do tablicy – otrzyma się tablicę z parami klucz-wartość

Przypisanie wartości pod nieistniejącym obiektem spowoduje dodanie wartości do hasza, np.:

p = {„lodówka” => 7}

p[„pralka”] = 8 >>

p = {„lodówka” => 7, „pralka” => 8}

.keys – pobiera z hasza listę wszystkich kluczy

.values – pobiera listę wartości

haszysz = {"1"=> "joda", "dwa" =>"sosna", 3=>"3"}

haszysz.each do |k, w|

puts "#{k}: #{w}"

end

kod, który zwraca to:

1: joda

dwa: sosna

3: 3

1. True i false

Są to wartości logiczne. Są to obiekty i używa się ich najczęściej w wyrażeniach warunkowych.

x = 1

x ==1

true

== operator porównania

x > 1

false

x >=1

true

!= operator sprawdzający, czy coś jest RÓŻNE

x != 1

false

Użycie operatorów porównania w praktyce.

x = 1

if x > 1

puts „x jest większe od 1’

else

puts „wcale nie większe”

end

>> wcale nie większe

OPERATOR NEGACJI - !

!true >> false

!false >> true

negować można dowolne obiekty

!1 >> false (Ruby uznaje 1 za prawdę)

!0>> false

oznacza to, że 0 jest prawdą. W Rubim wszystko jest prawdą poza false i nil

pusty string jest w Rubim prawdą

To różnica pomiędzy Rubim i innymi językami.

Inne operatory logiczne:

|| lub

&& i

1. Stałe

Stałe nie są obiektami (tak jak i zmienne), ale jedynie wskazują na obiekt. Stałych używa się wtedy, gdy nie chce się zmieniać obiektów, na które stała wskazuje.

Stałą tworzy się z wielkiej litery!

Bum = 2

BUM = 8 – też stała

Stałą da się zmienić, ale program wyświetla komunikat, że ją zmieniamy.

Stałe tworzy się po to, by nie zmieniać ich wartości.

Ruby jest dynamiczny, więc pozwala zmienić wartości stałych.

Gdy w nazwie stałej wszystkie litery są wielkie, to jest to klasyczna stała.

Gdy wielka jest tylko pierwsza litera, to według konwencji jest to klasa, czyli klasy w Rubim też są stałymi.

1. Przedziały

Przedział to struktura danych, która przechowuje zbiór wartości, który ma początek i koniec. Przedział tworzy się za pomocą operatora dwóch kropek, np.

x = 1..9 >>

1..9

Przedział to klasa „range”.

Operator jednej kropki ma wyższy priorytet niż operator dwóch kropek!

Aby narzucić Rubiemy priorytet używa się nawiasów.

1..10.class >> wyskoczy błąd, bo jedna kropka ma wyższy priorytet, czyli Ruby najpierw chce podać klasę dla 10, a potem zrobić z tego przedział.

(1..10).class >> teraz poda klasę, bo nawias zmienił priorytet.

Przydatne metody dla zakresów:

.begin – początek przedziału

.end – koniec przedziału

.include?(1) – sprawdza, czy 1 należy do przedziału

Definicja przedziału prawostronnie otwartego (oznacza to, że ostatnia liczba nie należy do przedziału):

u = 1…7

7 nie należy do przedziału

Ułamki należą do przedziału.

Przedział można skonwertować do tablicy za pomocą metody „.to\_a”

np.

u.to\_a

Przedział można iterować (wypisać wszystkie jego elementa).

u.each { |e| puts e}

1

2

3

4

5

6

7

Przedziały tworzy się także za pomocą liter!

„a”..”c”

(„a”..”c”).to\_a – konwertowanie do tablicy

[„a”,

„b”,

„c”]

Można generować dowolne kombinacje, np.

(„aa”..”bb”).to\_a

1. Wyrażenia

Nietypowe przypisanie:

a = b = 2

a >> 2

b >> 2

Nadawanie wartości wielu zmiennym:

a, b = 2, 1

a = 2

b = 1

Inne, ciekawe

a = [1, 2, 3, 4, 5]

b, c, \*d = a

b >> 1

c >> 2

d >> [3,4,5]

Ruby stara się dopasować schematy. Do „b” przypisał 1 element z tablicy, do „c” 2, a do d wszystkie pozostałe, bo operator gwiazdki „\*” przypisuje resztę do zmiennej, czy wartości, albo dokonuje ekspansji danej wartości.

Inny sposób użycia „\*”

x = 1..3

x.to\_a >> [1, 2, 3]

można to jednak zrobić tak:

[\*x] >>

[1,2,3]

„\*” dokonuje ekspansji wartości x

Łańcuchowe wywołanie wyrażeń:

czyli zamiast pojedynczego wykonywania metod:

[1,1,7,9,8,7,6,99, 7, 99, 101].uniq

potem wynik .sort

potem wynik .reverst

można po prostu zrobić

[zawartość tabeli].uniq.sort.reverse

1. Wyrażenia regularne

Wyrażenia regularne pozwalają: sprawdzić, czy tekst zawarty w stringu pasuje do jakiegoś schematu (np. czy string zawiera poprawny adres, lub numer telefonu) i wyszukuje w tekście fragmenty spełniające jakiś wzorzec.

Strona rubular.com

W kursie nie będzie mówił wiele o wyrażeniach regularnych.

„Wyrażenia regularne są bardzo interesujące i bardzo potężne”.

Wyrażenie regularne w Rubim tworzy się umieszczając je pomiędzy ukośnikami:

/test/

To jest 123 test (454) (2) 444 test (2312) test

\((\d+)\) – dopasowuje wszystkie liczby w nawiasach

\d – dopasowuje cyfry

(\d+) - dopasowuje cyfry i wypisuje dopasowania

\((\d+)\) – dopasowuje cyfry w nawiasach („\” sprawia, że Ruby nie uznaje nawiasu za komendę)

regex – regular expression

na steonie rubulat.com jest spis wyrażeń regularnych i można tam wpisywać tekst i ćwiczyć wyrażenia regularne

wyrażenie regularne jest klasy regexp

r = /\((\d+)\)/ (używa się 2 razy „/”, bo tak się tworzy wyrażenia regularne w Rubim.

tkt = „To jest 123 test (454) (2) 444 test (2312) test”

tkt = ~r (~ to operator dopasowania)

tkt = ~r = sprawdza, czy tekst w tkt pasuje do wzorca wyrażenia regularnego (zawartego w „r”) (czy zawiera liczbę w nawiasie)

>> 17

pokazuje numer znaku, w którym po raz pierwszy występuje pasujące dopasowanie

Gdy operator nie znajdzie dopasowania, zwróci „nil”

Aby program pokazał wszystkie dopasowania, trzeba wpisać:

tkt.scan(r) >>

[[„454”], [„2”], [„2312”]]

Wyrażenia regularne są używane w każdym projekcie, który pracuje z tekstem.

1. Wyrażenia warunkowe

Wykonują blok kodu, gdy spełniony jest określony warunek.

if – najpopularniejsze wyrażenie warunkowe

if

#kod

end

- jeśli warunek nie jest spełniony, to blok jest pomijany, Ruby przechodzi do wykonania kodu po słowie end

if x < 5

puts „X jest za małe”

end

if – else – wykonuje kod pod warunkiem spełnienia i niespełnienia

if warunek

#kod

else

#kod

end

if – elsif – else – wykonuje kod, gdy potrzebujemy więcej warunków. Gdy warunek będzie spełniony, Ruby go wykona. Gdy nie, zobaczy czy inny warunek jest spełniony. Jeśli jest, to go wykona. Jeśli nie, szuka dalej. Gdy nic nie będzie spełnione, wykona blok else.

if warunek

#kod

elsif inny\_warunek

#kod

elsif jeszcze\_inny\_warunek

#kod

else

#kod

end

ZŁOŻONE WARUNKI

x =

if x >= 5 && x < 10

puts "x jest w sam raz (5, 6, 7, 8 lub 9)"

end

if x < -5 || x > 5

puts "Wartość bezwględna jest spoko."

end

blok wewnątrz każdego if zostanie spełniony, gdy warunek zostanie spełniony.

Spełnienie 1 warunku – pojawi się jedna treść.

Dwóch – dwie

Ani jednego - nie pojawi się nic.

Operator trójargumentowy – rzadko spotykane, ale pozwala na wykonanie typowego „if – else” w jednej linijce kodu.

warunek ? jeśli\_prawda : jeśli\_fałsz

x = 11

x > 5 ? puts("Większe od 5") : puts("Mniejsze od 5")

if z zaprzeczeniem

unless – instrukcja warunkowa będąca zaprzeczeniem „if”. If zastępujemy słowem „unless”. Gdy warunek nie jest spełniony, blok jest wykonywany.

x = 11

unless x < 0

puts "tak"

end

unless – else - tak, jak z if. Nie jest to polecane, trudno zrozumieć.

Modyfikator wyrażeń – używa się, gdy chce się wykonać prostą instrukcję pod warunkiem

składnia:

instrukcja if warunek

instrukcja unless warunek

x = -2

signum = if x < 0

-1

elsif x == 0

0

elsif x > 1

end

puts signum

signum lub cokolwiek innego przypisuje wartość. Bez tego się nic nie wyświetli (kod się nie wykona).

W Rubim wszystko jest wyrażeniem, instrukcja warunkowa także, więc zwraca wartość.

1. Instrukcja case

Gdy jest wiele warunków, instrukcja if przestaje się skalować, gdyż jest nieczytelna. W tym wypadku używamy instrukcji case, czyli tzw. instrukcji wyboru. Składnia case:

case

when warunek

#....

when warunek

#....

else

#...

end

Case – tak jak if – posiada else. Else jest opcjonalny.

Częściej występujący wariant case:

case wyrażenie

when wartość1

#...

when wartość2

#...

else

#...

end

Przykład użycia pierwszego wariantu:

x = 0

case

when x == 1

puts "x jest jedynką"

when x > 2

puts "x jest większe od dwóch"

else

puts "inna wartość"

end

Drugi wariant:

z = 101

case z

when 1

puts "z jest jedynką"

when 10..19

puts "z jest równe kilkanaście"

when 1..10

puts "z jest w pierwszej dziesiątce"

when 20..100

puts "z jest to kilkadziesiąt"

else puts "z to kilkaset"

end

może być takie when też:

when 1,2,3,4,5,6

CASE WYWOŁUJE TYLKO PIERWSZY BLOK SPEŁNIAJĄCY WARUNEK I NIE SPRAWDZA DALSZYCH!

Można też przypisać do zmiennej i wykonać tę zmienną:

z = 1

bam = case z

when 1

"jest 1"

when 10..19

"jest kilkanaście"

when 1..10

"jest w dziesiątce"

when 20..100

"kilkadziesiąt"

else

"kilkaset"

end

puts bam

1. Pętle

To też struktury kontrolne.

Służą do cyklicznego wywoływania grupy instrukcji.

loop – pętla nieskończona

loop do

#...

end

Kod zawarty pomiędzy loop i end będzie wykonywany w nieskończoność.

Pomimo to, z tej pętli można wyjść, kontrolować ją.

Opuszczanie pętli/iteracji

break – wychodzi z pętli

next – przerywa iterację pętli, przechodzi do następnej

retry – przerywa iterację i ponawia jej wykonanie

redo – powtarza całą pętlę

1.x = 0

2. loop do

3. x += 1

4. if x.even?

5. next

6.end

7.if x > 9

8. break

9.end

10.puts x

11. end

Rozumiem to tak:

3. x = 0 +1 = 1

4 i 5. program pomija x nieparzyste

7. przerywa pętlę

10. wyświetla wszystkie x spełniające warunek

11. po co end?

Można to zapisać inaczej – z modyfikatorami wyrażeń „if”:

x = 1

loop do

x += 1

next if x.odd?

break if x > 10

puts x

end

While – inny rodzaj pętli

wykonuje blok tak długo, jak długo spełniony jest warunek

while warunek

#…

end

Najpierw jest spełniany warunek, a dopiero potem wykonywany jest blok, zatem w momencie, gdy Ruby wchodzi w pętlę i widzi, że warunek nie jest spełniony, to nie wykonuje ani razu warunku.

Tak, jak „if” ma „unless” (przeciwieństwo), tak i „while” ma przeciwieństwo, czyli „until”

until warunek

#...

end

Until to pętla będąca zaprzeczeniem (odwrotnością) while, czyli wykonuje kod wewnątrz pętli dopóki warunek NIE JEST SPEŁNIONY. Gdy warunek jest spełniony, Ruby opuszcza pętle i wykonuje dalszy kod.

While i untli mogą być modyfikatorami wyrażeń.

Ciekawy kod:

until (text = gets.chomp) == "exit"

puts "#{text}"

end

wyświetla to, co wpisuje użytkownik, dopóki nie wpisze on „exit”. Jak wpisze, Ruby kończy pętlę.

x = 0

while x < 5

puts "buch"

x += 1

end

buch

buch

buch

buch

buch

Ale nie jest to lubiany przez programistów kod. Lepiej używać do tego iteratora times.

5.times do

puts „legiunia”

end

legiunia

legiunia

legiunia

legiunia

legiunia

Każda liczba w Rubim posiada metodę „times” i służy ona do wykonania danego bloku tyle razy, ile każe times.

Bloki można też wykonać za pomocą składni z nawiasami klamrowymi.

5.times do { puts „ziomek” }

W przypadku „Times” zmienna także potrafi liczyć iterację:

5.times do |x|

puts "daj pieniądze: #{x}"

end

daj pieniądze: 0

daj pieniądze: 1

itd.

Iteracja po kolekcjach:

["dziki" , "zielony" , "kapusta"].each do |animal|

puts animal

end

dziki

zielony

kapusta

Aby w bloku był zarówno obiekt, jak i indeks, pod którym obiekt jest w tablicy, można użyć metody „.each\_with\_index”

Inne iteratory pozwalające na iterację „od-do”:

1. upto

3.upto(8) { |z| puts z }

3

4

5

6

7

8

1. downto (odwrotność upto)

8.downto(3) { |j| puts j }

8

7

6

5

4

3

1. step – iterowanie w regularnych odstępach

4.step(15, 3) {|i| puts i}

4

7

10

13

1. Wyjątki

To struktury kontrolne służące do obsługi sytuacji wyjątkowych, czyli głównie błędów, np. odczytywany plik nie istnieje na dysku, albo połączenie internetowe zostało zerwane, albo podzielono przez 0.

Biblioteki potrafią rozpoznać takie sytuacje dzięki wyjątkom odpowiedniego rodzaju.

puts "Początek"

a = 1

b = 0

puts a/b

puts "Koniec"

wyświelta się „Początek”, ale „Koniec” już nie. Dzieje się tak dlatego, że gdy pojawia się błąd, przerywa się działanie programu.

Błąd jest nieobsłużonym wyjątkiem. Gdy wystąpi, program kończy pracę w miejscu wystąpienia wyjątku. Dlatego właśnie programy muszą obsługiwać wyjątki.

puts "Początek"

a = 1

b = 0

begin

puts a/b

rescue

puts "Błąd"

end

puts "Koniec"

Wyświetla się:

Początek

Błąd

Koniec

Następuje jednak taki problem, że „Błąd” wyskoczy za każdym razem, gdy program nie będzie w stanie wykonać polecenia, a chodziło o zdefiniowanie konkretnego wyjątku dzielenia przez 0.

puts "Początek"

a = 1

b = 0

begin

puts a/bd

rescue => d

puts "Błąd: #{d.class}" – przypisujemy do rescue zmienną(?) d (albo jakąkolwiek), a następnie sprawdzamy, jaki to rodzaj błędu. Okazuje się, że jest to „NameError” bo zmienna „bd” nie została zdefiniowana.

end

puts "Koniec"

puts "Początek"

a = 1

b = 0

begin

puts a/b

rescue ZeroDivisionError

puts "Nie dziel chamie przez zero"

end

puts "Koniec"

Teraz błąd jest obsłużony prawidłowo.

Ale już ten program nie zostanie wykonany:

puts "Początek"

a = 1

b = 0

begin

puts a/b

rescue ZeroDivisionError

puts "Nie dziel chamie przez zero"

end

Można obsłużyć kilka rodzajów błędów poprzez kaskadowe wywołanie „Rescue”:

puts "Początek"

a = 1

b = 0

begin

puts a/baaa

rescue ZeroDivisionError

puts "Nie dziel chamie przez zero"

rescue NameError

puts "Nie zdefiniowałeś zmiennej jełopie"

end

puts "Koniec"

POWINNO SIĘ OBSŁUGIWAĆ WYJĄTEK PODAJĄC KONKRETNĄ NAZWĘ BŁĘDU, CZYLI TAK:

RESCUE ZERODIVISION ERROR

Program można wywołać niezależnie od tego, czy błąd wystąpił używając komendy „ensure":

puts "Początek"

a = 1

b = 0

begin

ensure

puts "Zawsze się wykona"

end

puts "Koniec"

Kod zostanie wykonany.

puts "Początek"

a = 1

b = 0

licznik = 0

begin

puts a/b

rescue ZeroDivisionError

licznik += 1

puts "Nie dziel przez zero"

retry if licznik < 4

else

puts "Brak błędu"

ensure

puts "Zawsze się wykona"

end

puts "Koniec"

Początek

Nie dziel przez zero

Nie dziel przez zero

Nie dziel przez zero

Nie dziel przez zero

Zawsze się wykona

Koniec

1. Metody, zwracanie wartości, argumenty

Definiowanie i wywoływanie metod.

Doraźna definicja metody to: metoda to nazwany zbiór instrukcji, który się tworzy, aby później je wywołać w różnych miejscach i przy różnych parametrach.

Tak się tworzy metodę:

def nazwa

puts „koks”

end

nazwa

Metody mogą posiadać argumenty, dzięki którym przekazujemy im nowe dane i można sterować ich działaniem. Argumenty podaje się w nawiasach.

def metoda(imię)

puts "Elo #{imię}"

end

metoda "Saturnin"

Elo Saturnin

Podczas definiowania metod powinno się używać nawiasów, ale podczas ich wywoływania nawiasy są opcjonalne.

Metoda, która równocześnie przyjmuje argumenty i zwraca wartość:

def podwójne(z)

z\*2

end

puts podwójne(1731)

3462

Metoda przyjmująca dwa argumenty:

def add(a, b)

a + b

end

puts add(98, 98)

196

1. Wartości domyślne argumentów

def greet(name, salutation)

puts "#{salutation} #{name}"

end

puts greet("Marek", "student")

student Marek

def greet(name, salutation=”welcome”) – przypisuje wartość domyślną do salutation, dzięki czemu, nie trzeba jej podawać w puts, a i tak wynik będzie: welcome Marek

puts "#{salutation} #{name}"

end

puts greet("Marek", "student")

student Marek

1. Argumenty nazwane

Wywołanie metody nie daje wiedzy o jej strukturze.

Gdy metoda posiada dużą liczbę argumentów, i nie wiadomo, co robi konkretny argument, warto używać argumentów nazwanych.

def greet(name, salutation: "Hey", exclamation:"!")

puts "#{salutation} #{name}#{exclamation}"

end

greet("Ruby")

Hey Ruby!

1. Dynamiczna liczba argumentów

Ruby pozwala definiować metody o zmiennej, dynamicznej liczbie argumentów.

Aby zdefiniować taką metodę używa się operatora \*, czyli „splat”.

def test(a, \*rest)

puts rest.class

puts rest.inspect

end

puts test(1,2,3,4)

array[2, 3, 4] – wynika z tego, że rest jest tablicą z tymi 3 wymienionymi argumentami. Program pokazuje jednocześnie, że rest jest tablicą (array) i jej zawartość

\*rest przechowuje listę wszystkich pozostałych argumentów niezależnie od ich liczby

ef greet(name1, name2, \*rest)

puts "Hello #{name1}"

puts "Hello #{name2}"

rest.each do |name|

puts "Hello #{name}"

end

end

greet("Ruby", "Python", "Java", "Java Script")

Hello Ruby

Hello Python

Hello Jaca

Hello Java Script

1. Bloki

Będziemy teraz definiować metody, które przyjmą bloki.

def twice – metoda wywołuje dwukrotnie blok, który się do niej przekazuje

yield – wywołuje blok przekazany do metody

yield

end

twice { puts "Hi, hello!"}

Hi, hello!

Hi, hello!

Teraz napiszemy metodę tak, by sprawdzić, czy blok został przekazany. Gdy będzie przekazany, zostanie wywołany, gdy nie, wyświetli się odpowiednia informacja.

def triple

if block\_given?

yield

yield

yield

else

puts "There is no block!!"

end

end

triple {puts "There is block, ha!"}

There is block, ha!

There is block, ha!

There is block, ha!

Dzięki tej metodzie można wywołać dowolną zmienną (na pewno stringa).

Teraz metoda „times”, która przekazuje argument do bloku:

def my\_times(n)

until n.zero?

yield(n)

n -= 1

end

end

my\_times(3) { puts "Replace!"}

Replace!

Replace!

Replace!

Wykorzystanie tego, że metoda przekazuje argument do bloku:

def my\_times(n)

until n.zero?

yield(n)

n -= 1

end

end

my\_times(3) { |x| puts "Replace #{x}"}

Replace 3

Replace 2

Replace 1

def test(&block) – za pomocą „&” deklaruje się blok w sposób jawny. Zmienna „block” będzie przechowywać blok.

puts block.class

end

test { puts "Hello"}

Proc – oznacza to, że zmienna „block” jest obiektem klasy „Proc”. To obiekt przechowujący kod przekazywany do bloku.

1. Proc i lambda

Proc to obiekt proceduralny.

def greet(what)

"Hello #{what}"

end

greet1 = proc { |what| "Hello #{what}"}

greet2 = lambda { |what| "Hello #{what}"}

puts greet("Mordo")

puts greet1.call("Pysku") – call wywołuje proca i lambdę

puts greet2.call("Ryju")

Hello Mordo

Hello Pysku

Hello Ryju

Wywołanie metody bez argumentu da błąd –

puts greet – błąd

Wywołanie proca bez argumentu zwróci „Hello”, co oznacza, że proc nie sprawdza, ile argumentów do niego przekazano. Można go wywołać z dowolną liczbą argumentów lub bez argumentu.

puts greet1.call(„japa”)

Hello japa

Lambdy nie da się wywołać bez argumentów. Lambda pilnuje liczby argumentów.

Kolejna różnica pomiędzy blokami a obiektami proceduralnymi:

a = 123

def test

puts a

end

test1 = lambda { puts a}

test2 = proc { puts a}

test1.call

test2.call

wywołanie samego “test” – program nie widzi zmiennej a, zwraca błąd.

test1.call daje 123

test2.call daje 123

proc oraz lambda widzą zmienną a

Metoda jest odizolowana od lokalnych zmiennych, wokół których została zdefiniowana.

Proc przechowuje blok kodu, z którym został zdefiniowany oraz kontekst wokół którego został zdefiniowany.

Definiując proca wrzuca się do niego zarówno kod, jak i wszystkie zmienne lokalne wokół których został zdefiniowany.

To bardzo istotna różnica pomiędzy metodami a procami.

double = proc { |y| y\*7} – proc zwraca siedem razy większą wartość argumentu doń przekazanego

puts double.call(11)

77

Blok przekazywany do metody to proc.

puts [1,2,3,4,5,6,7].map { |q| q\*2}

2, 4, 6, 8, 10, 12, 14

double = proc { |y| y\*8}

puts [1,2,3,4,5,6,7].map(&double) – jawne przekazanie proca

8, 16, 24, 32 itd

Było to przekazywanie bloku do metod.

1. Klasy

Różnica pomiędzy klasą a obiektem:

obiekt (instancja) to egzemplarz klasy, klasa zaś to opis jakiegoś rodzaju (zbiór?).

class Person

end

o1 = Person.new

o2 = Person.new

puts o1.object\_id

puts o2.object\_id

puts o1 == o2

70134966761820 – czy to informuje o zapisaniu danego obiektu w jakimś bicie?

70134966761800

false

class Person

def greet

puts "Hello!"

end

end

o1 = Person.new

o2 = Person.new

o1.greet

o2.greet

Hello

Hello

class Person

def greet

puts "Hello!"

end

def greeting

puts "Hi!"

end

end

o1 = Person.new

o2 = Person.new

o1.greet

o2.greeting

Hello

Hi!

class Person

def greet

puts greeting

end

def greeting

puts "Hi! I'm #{object\_id}"

end

end

o1 = Person.new

o2 = Person.new

o1.greet

o2.greeting

Wywołując #{object\_id} wewnątrz klasy nie trzeba pisać, do którego obiektu się odwołujemy. Przy pracy z obiektem, każde wywołanie metody jest wywoływane W KONTEKŚCIE KONKRETNEGO OBIEKTU.

„Self” w Rubim jest słowem kluczowym i oznacza „samego siebie”. Dla przykładu: wewnątrz klasy „Person”, „selfem” jest klasa „Person”. Wewnątrz metody dla obiektów, „self” jest obiektem, na którym wywołuje się metodę.

class Person

def greet

puts self.greeting

end

def greeting

puts "Hi! I'm #{self.object\_id}"

end

end

o1 = Person.new

o2 = Person.new

o1.greet

o2.greeting

W większości przypadków, „self” jest automatycznie wpisywany przez Rubiego, bo wiadomo, że w momencie wywoływania np. metody, jest ona wywoływana w kontekście samej siebie, a nie w kontekście innego obiektu. Oznacza to, że używanie „selfa” nie jest wszędzie obowiązkowe.

1. Zmienne instancji i atrybuty

Zmienną instancji (obiektu) można zdefiniować tak, jak i zmienną lokalną. Zmienna instancji jest deklarowana dla konkretnego obiektu i dla każdego obiektu tej samej klasy może posiadać inną wartość, bo każdy z obiektów posiada odwołanie do własnej zmiennej instancji.

class Person

def greet

puts @greeting

end

def make\_greeting

@greeting = "Hello " #zmienna instancji rozpoczyna się od znaku "@"

#nawet gdybyśmy zdefiniowali zmienną lokalną o tej samej nazwie - greeting = "Different hello"

# zmienna instancji jest przechowywana i istnieje dopóki istnieje obiekt

greeting = "Inne witaj"

end

end

o1 = Person.new

o2 = Person.new

o2.make\_greeting

o1.greet

o2.greet

Hello

Wynika z tego następujący wniosek: nawet jeśli zmienna instancji jest niezdefiniowana (niewyłowywana), to ona nadal istnieje, tylko ma wartość „nil”. Metoda instancji jest przechowywana wewnątrz obiektu i nikt z zewnątrz nie ma do tej zmiennej dostępu. Gdy chce się w danym obiekcie coś przechować, ale tak, by inni nie mogli tego modyfikować, to należy przechować to w zmiennej instancji, czyli w zmiennej należącej wyłącznie do danego obiektu.

Istnieje możliwość pośredniego dostępu do tej zmiennej:

class Person

def greet

puts @greeting

end

def set\_greeting(new\_greeting)

@greeting = new\_greeting

end

end

o1 = Person.new

o2 = Person.new

o2.set\_greeting("Hi")

o2.greet

#kod ten w sposób pośredni zwraca zmienną instancyjną @greeting

Można napisać metodę zwracającą wartość zmiennej instancyjnej @greeting

class Person

def greet

puts @greeting

end

def set\_greeting(new\_greeting)

@greeting = new\_greeting

end

def get\_greeting

@greeting

end

end

o1 = Person.new

o2 = Person.new

o2.set\_greeting("Hi")

o2.greet

puts o2.get\_greeting

class Person

def greet

puts @greeting

end

def set\_greeting(new\_greeting)

@greeting = new\_greeting

end

def get\_greeting

@greeting

end

end

o1 = Person.new

o2 = Person.new

o2.set\_greeting("Hi")

o2.greet

puts o2.get\_greeting

Hi

Hi

#bez linijaki 20 i 22 nie będzie wywołany kod z linijki 23, bo get\_greeting jest skorelowany z set\_greeting i greet i muszą one być wywołane.

W programowaniu obiektowym zdarza się, że musi być możliwość ustalenia lub dostępu do zmiennej instancyjnej z zewnątrz. Tak to wygląda w odpowiedni sposób w Rubim. Tego typu metody w Rubim nazywa się geterami lub seterami:

class Person

def greet

puts @greeting

end

def greeting=(new\_greeting)

@greeting = new\_greeting

end

def greeting

@greeting

end

end

o1 = Person.new

o2 = Person.new

o2.greeting = "Yo!"

o2.greet

puts o2.greeting

# wygląda to ta, jakby się ustawiało zmienną wewnątrz obiektu o2, ale tak nie jest, a tak naprawdę wywołuje się metodę.

#wygląda to tak, jakby metoda "greeting = " była przypisaniem, ale tak nie jest. To po prostu wywołanie metody "greeting ="

# linijka 22 wygląda jak odczytanie wartości, ale tak nie jest. Jest to po prostu wywołanie metody, która odczytuje wartość.

Metody greeting= i greeting to jedne z najpopularniejszych rodzajów metod w programowaniu obiektowym, dlatego Ruby udostępnia skrót, który to samo, tylko w jednej linijce.

class Person

attr\_accessor :greeting – to jest atrybut klasy. Gdy mowa o atrybucie, oznacza to, że istnieje jakaś zmienna instancyjna, która przechowuje ten atrybut i są metody, które służą do dostępu do tego atrybutu, czyli setery i getery

(to zastępuje te dwie metody: #def greeting=(new\_greeting)

#@greeting = new\_greeting

#end

#geter wygląda tak:

# def greeting

# @greeting

#end)

def greet

puts @greeting

end

end

o1 = Person.new

o2 = Person.new

o2.greeting = "Welcome"

o2.greet

puts o2.greeting

Welcome

Welcome

Istnieją trzy rodzaje takich skrótów w Rubim:

attr\_accesor: definiuje setera i geterta

attr\_reader: – umożliwia odczytanie atrybutu, ale uniemożliwia zapis

attr\_writer: - umożliwia zapisywanie dodawanej zmiennej bez możliwości odczytania jej

attr\_accesor to tak naprawdę połączony attr\_reader i attr\_writer

więc można zapisać go tak:

attr\_reader :greeting

atte\_writer :greeting

Należy skorzystać ze skrótu wszędzie tam, gdzie używa się atrybutów. Umożliwia to skrócenie kodu i jest szybsze.

1. Konstruktor

Każda klasa w programowaniu obiektowym posiada specjalną metodę nazywaną „konstruktorem”. Jest on wywoływany w momencie tworzenia obiektu. Zawsze, gdy wywołuje się metodę „.new”, wywołuje się konstruktor.

class Person

attr\_accessor :greeting

def initialize #metoda "initialize" definiuje własny konstruktor

puts "I'm in the constructor"

end

def greet

puts @greeting

end

end

o2 = Person.new # konstruktor jest wywoływany w momencie tworzenia obiektu, czyli w tym miejscu

#wywołując metodę ".new" Ruby alokuje pamięć dla tego obiektu, a następnie wywołuje konstruktor

o2.greeting = "Yo!"

o2.greet

puts o2.greeting

I’m in the constructor

Yo!

Yo!

W konstruktorze można ustalać zmienne instancji:

class Person

attr\_accessor :greeting

def initialize(name)

puts "I'm in the constructor"

@name = name

end

def greet

puts @greeting

end

end

o2 = Person.new("Ruby") #ten argument w nawiasie musi być, bowiem konstruktor wywołuje metodę z argumentem

o2.greeting = "Yo!"

o2.greet

puts o2.greeting

#tak oto stworzyliśmy obiekt z dodatkową zmienną instancji, która się nazywa @name i ma wartość "Ruby"

Konstruktor służy najczęściej do nadania początkowej wartości, albo do przygotowania obiektu do dalszego działania.

(W Rubim nie zarządza się pamięcią. Ruby sam to robi.)

1. Dziedziczenie, instrukcja super

Dziedziczenie jest często wykorzystywane w programowaniu obiektowym. Umożliwia ono stworzenie klasy podrzędnej, która dziedziczy wszystkie funkcjonalności z klasy nadrzędnej. Najczęściej klasa dziedzicząca jest bardziej szczegółowa.

Celem dziedziczenia jest zmodyfikowanie pewnych metod.

class Person

attr\_accessor :greeting

def initialize(name)

puts "I'm in the constructor"

@name = name

end

def greet

puts @greeting

end

end

class LoudPerson < Person

def greet

puts @greeting.upcase

end

end

o2 = LoudPerson.new("Ruby")

o2.greeting = "Yo!"

o2.greet

I’m in the constructor.

YO!

class Person

attr\_accessor :greeting

def initialize(name)

puts "I'm in the constructor"

@name = name

end

def greet

puts @greeting

end

end

class LoudPerson < Person

def greet

puts @greeting.upcase

end

end

o1 = Person.new("janusz")

o1.greeting = "Welcome"

o1.greet

o2 = LoudPerson.new("Ruby")

o2.greeting = "Yo!"

o2.greet

I’m in the constructor

Welcome

I’m in the constructor

YO!