МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НТУ «ДНІПРОВСЬКА ПОЛІТЕХНІКА»

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра інформаційних технологій та комп'ютерної інженерії

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

до виконання лабораторних робіт з дисципліни
«Технології розподілених систем та паралельних обчислень»
на мові Ruby

для студентів спеціальності
126 «Інформаційні системи та технології»

Дніпро
НТУ «ДП»
2020

Методичні рекомендації до виконання лабораторних робіт з дисципліни «Технології розподілених систем та паралельних обчислень» на мові Ruby для студентів спеціальності 126 «Інформаційні системи та технології» / Г.М. Коротенко, Л.М. Коротенко. - Д.: НТУ «Дніпровська політехніка», 2020. – 95 с.

Методичні рекомендації містять відомості про особливості мови програмування Ruby, яка є інтерпретованою, повністю об'єктно-орієнтованою з чіткою динамічною типізацією, а також з відповідністю функційній, імперативній і логічній парадигмам. Розглянуто застосування цієї мови для вирішення задач в галузі розподілених та паралельних обчислень.

Призначено для студентів факультету інформаційних технологій, які навчаються за спеціальністю 126 «Інформаційні системи та технології».

Погоджено рішенням науково-методичної комісії спеціальності 126 Інформаційні системи та технології (протокол № 7 від 27.08.2021).

ЗМІСТ

ЛАБОРАТОРНА РОБОТА №1. Початок роботи з Ruby. Основні типи даних, змінні і математичні функції. …… 6
1.1. Початок роботи з інтерпретаторами Ruby…… 6
1.2. Підготовка та виконання програм на мові Ruby…….. 9
1.3. Коментарі в програмах……..…….. 10
1.4. Виведення даних на екран……..……. 10
1.5. Формування імен змінних……..……. 11
1.6. Базові типи даних……..……..……. 11
1.7. Представлення чисел……..……. 11
1.8. Перетворення рядка тексту в число……..…… 12
1.9. Використання математичних функцій…… 14
1.10. Оператори Ruby……..……..…… 15
1.11. Завдання та вміст звіту до даної лабораторної роботи…… 15
ЛАБОРАТОРНА РОБОТА №2.Особливості роботи з чисельними типами даних……..……..…… 16
2.1. Використання базових типів даних2.2. Об'єкти і методи в Ruby…. 16
2.2. Об'єкти і методи в Ruby……..……..…… 17
2.3. Простір об'єктів……..……..……. 18
2.4. Типи числових даних, форма запису чисел, присвоювання значень і зміст змінних (об'єктів) ……..…….. 19
2.5. Представлення числових даних в мові Ruby……. 20
2.6. Правила представлення числових даних в мові Ruby…… 22
2.7. Створення даних заданих типів і перетворення чисел…… 23
2.8. Арифметичні операції з цілими і дробовими числами…… 25
2.9. Отримання випадкових чисел……..…… 26
2.10 Не математичні функції для перетворення даних…….. 26
2.11. Завдання до лабораторної роботи……..…… 27
ЛАБОРАТОРНА РОБОТА №3. Керуючі структури мови Ruby. (Умовні оператори, цикли і методи-ітератори). ……..…… 28
3.1. Отримання і використання логічних значень……. 28
3.2. Порівняння логічних значень……..……. 32
3.3. Формування рядків при друкуванні…… 34
3.4. Оператори розгалуження……..…….. 34
3.5. Оператори циклу і методи-ітератори……..…… 37

3.6. Блоки циклів і ітераторів……..……..…… 37
3.7. Оператор циклу з параметром for..do……. 38
3.8. Оператор циклу з передумовою while. …….. 39
3.9. Оператор циклу з умовою поста (while або until) …….. 39
3.10. Використання циклів while та until в якості модифікаторів…… 40
3.11. Цикл loop……..……..……. 40
3.12. Методи-ітератори each, each line, times, upto і step…… 41
3.13. Завдання до лабораторної роботи……..…… 44
ЛАБОРАТОРНА РОБОТА №4. Масиви, вектори і матриці…… 45
4.1. Опис і заповнення одновимірних масивів…….. 45
4.2. Приклади використання масивів……..……. 46
4.3. Застосування циклів і ітераторів для обробки масивів…… 47
4.4. Використання методів для роботи з масивами…….. 49
4.5. Створення та заповненння багатовимірних масивів…… 53
4.6. Створення і заповнення двовимірних (n × m) масивів…… 53
4.7. Сумування, добуток, і агрегація елементів за допомогою ітераторів……..……..…….. 55
4.8. Використання бібліотек mathn та matrix……. 56
4.9. Обчислення добутку двох матриць, оформлення у вигляді методу matrix_product……..……..…….. 57
4.10. Завдання до лабораторної роботи……..…… 58
ЛАБОРАТОРНА РОБОТА №5. Методи та блоки програми…….. 63
5.1. Вбудовані методи мови Ruby……..……. 63
5.2. Методи - глобальні функції……..…… 63
5.3. Методи - оператори……..……..…… 64
5.4. Методи класу Numeric і Сінглтон-методи……. 64
5.5. Методи-ітератори……..……..…… 65
5.6. Взаємозв'язок методів і їх використання……..…… 65
5.7. Блоки та їх використання……..…….. 69
5.8. Особливості ітераторів при використанні блоків…… 71
5.9. Числові ітератори……..……..…… 72
5.10. Методи……..……..……. 72
5.11. Обчислення визначеного інтеграла……..…… 73
5.12. Обчислення дисперсії і стандартного відхилення…… 73
5.13. Завдання до лабораторної роботи……..…… 74

ЛАБОРАТОРНА РОБОТА №6. Основи ООП в Ruby…….. 75
6.1. Об’єкти……..……..…….. 75
6.2. Вбудовані класи……..……..…… 75
6.3. Модулі та класи-домішки……..……. 75
6.4. Створення класів……..……..…… 77
6.5. Методи та атрибути……..……..…… 79
6.6. Кодування під час виконання……..……. 79
6.7. Наслідування……..……..…….. 80
6.8. Завдання до лабораторної роботи……..…… 82
ЛАБОРАТОРНА РОБОТА №7. Багатопоточність в Ruby…… 83
7.1. Створення потоків Ruby……..……. 83
7.2. Управління потоками……..…….. 84
7.3. Змінні потоку……..……..…….. 84
7.4. Потоки і виключення……..……..…… 85
7.5. Управління планувальником потоків……..…… 86
7.6. Завдання до лабораторної роботи……..……. 88
ЛАБОРАТОРНА РОБОТА №8. Використання потоків в Ruby.
Паралельне та розподілене програмування. ……. 89
8.1. Потоки і обчислювальні операції……..…… 89
8.2. Паралелізм на рівні процесів……..…….. 90
8.3. Застосування модуля DRb для синхронізації процесів…… 91
8.4. Паралелізм в Ruby on Rails……..…… 91
8.5. Завдання до лабораторної роботи……..……. 92
Список використаних джерел…….. 93

ЛАБОРАТОРНА РОБОТА №1.

Початок роботи з Ruby. Основні типи даних, змінні і математичні функції.

Тема: Рішення простих математичних задач.

Мета: Використання інтерпретаторів Ruby. Завдання імен змінних, обробка різних простих типів даних і застосування основних математичних функцій. Введення і виведення даних з використанням екрану.

Теоретичні відомості:

1.1. Початок роботи з інтерпретаторами Ruby

Програми на мові Ruby це звичайні текстові файли з ASCII-кодами операторів мови, які не підлягають компіляції. Для їх виконання потрібен інтерпретатор. Як правило, щоб запустити програму на мові Ruby, необхідно викликати інтерпретатор і передати йому ім'я файлу з програмою в якості параметра, або набрати в рядку інтерпретатора (наприклад, irb) відповідні набори операторів (команд) мови.

При встановленому на комп'ютері комплексі (дистрибутиві) програм Ruby програмістом можуть бути відкриті і використовуватися для роботи два вікна інтерпретаторів з різною функціональністю.

1. Перше вікно називається «Start Command Prompt with Ruby». Дане вікно представляє доступ до інтерпретатора під назвою «Командний рядок з Ruby» (рис. 2.1). На його вхід подаються імена програм, що містять оператори мови Ruby. Увага: обов'язковою умовою виклику і обробки програми є «підведення» (вказання шляху) інтерпретатора до каталогу, в якому розташовується дана програма.

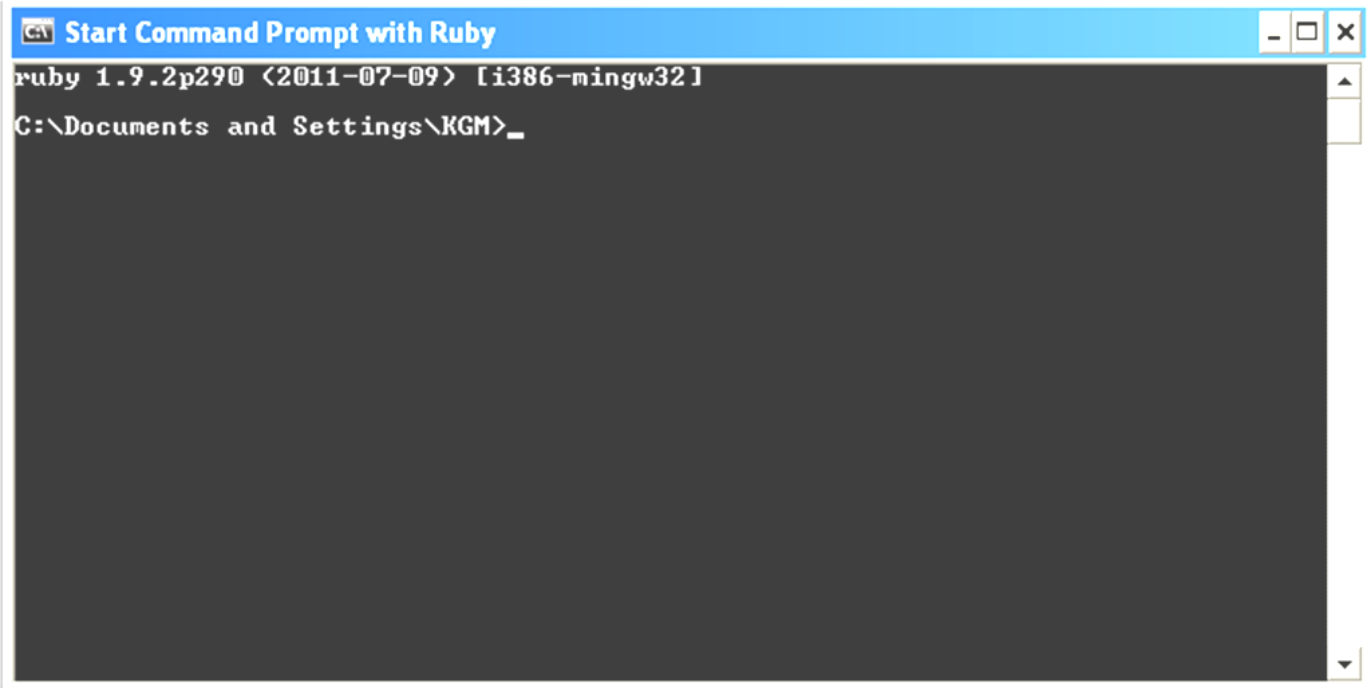


Рис.1.1 Вікно інтерпретатора програм на мові Ruby - «Командний рядок з Ruby». У вікні вказано шлях до каталогу (папки, директорії) з ім'ям KGM.

2. Друге вікно називається «Interactive Ruby» (Інтерактивний інтерпретатор Ruby), а сам інтерпретатор носить ім'я - «irb» (рис. 1.2). На вхід даного інтерпретатора зазвичай подаються (вводяться) оператори мови Ruby.

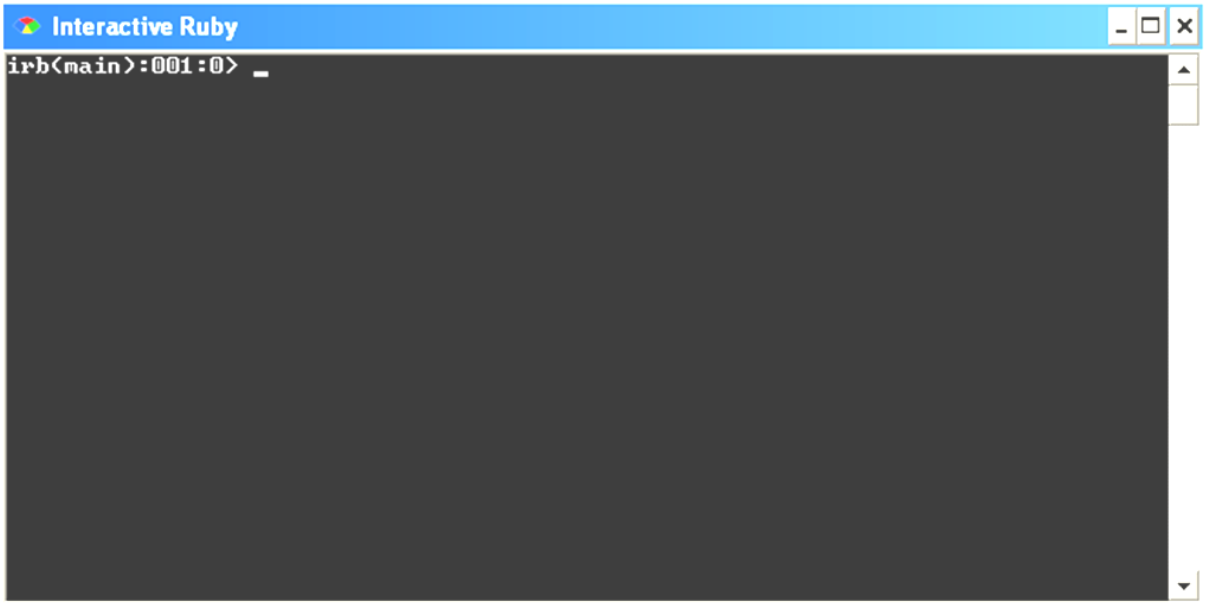


Рис. 1.2. Вікно інтерпретатора операторів мови Ruby – «Інтерактивний інтерпретатор Ruby» (irb)

Дане вікно, яке називають «Інтерпретатор Ruby» або «Інтерактивний Ruby» представляє собою інтерактивну оболонку для виконання синтаксичних конструкцій мови Ruby – Interactive Ruby Shell (irb).

Даний інтерактивний засіб дозволяє виконувати, як окремі оператори коду Ruby, так і цілі конструктивні блоки (умовні оператори, блоки програм, цикли та ін.), а крім того, і завершення програми, оформлені у вигляді файлів з розширенням (.rb).

Як приклад введемо в вікні інтерпретатора «Інтерактивний Ruby» оператор обчислення косинуса кута 90º, що відповідає значенню π/2 радіан. Результат взаємодії користувача і інтерпретатора представлений нижче.

```
irb(main):004:0> Math.cos(3.14159265/2)
=> 1.7948965149187836e-09
```

Результат обчислення приблизно дорівнює нулю (близький до нуля) і буде тим ближче до нуля, чим точніше буде задано значення числа π (Пі).

Зверніть увагу, що не тільки функція cos(), але і всі інші стандартні математичні функції в мові Ruby є методами класу Math. Більш детально клас Math буде розглянуто далі.

Важливо те, що інтерпретатор команд «Інтерактивний інтерпретатор Ruby» (irb) може бути викликаний з інтерпретатора «Командний рядок Ruby» шляхом виклику команди: irb. Далі робота ведеться в інтерпретаційному інтерактивному режимі. Вихід з цього режиму здійснюється викликом команди: exit (див. рис. 1.3).

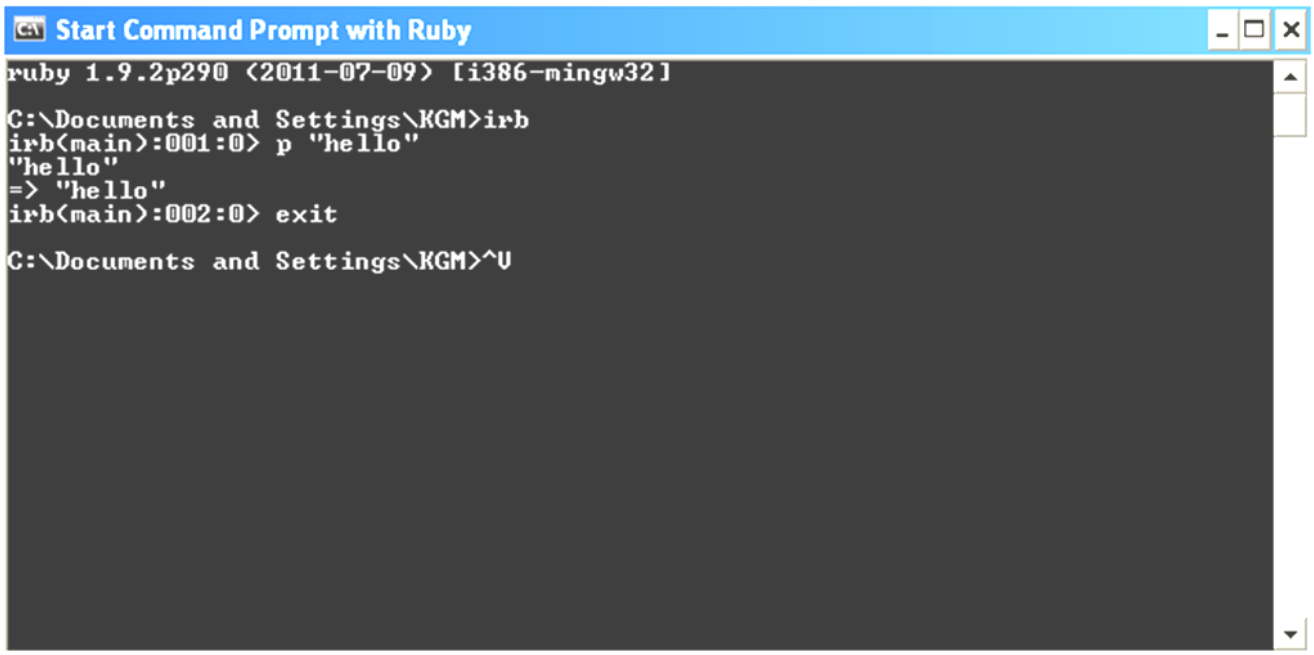


Рис. 1.3. Виклик інтерактивного інтерпретатора мови Ruby - irb з вікна інтерпретатора «Командний рядок з Ruby».

У рядку інтерпретатора irb введений оператор «p "hello"», який забезпечує вивід на екран текстового рядки "hello". Особливості роботи даного оператора розглядаються далі.

На рис.1.4 наведено виклик оператора «p" hello "» безпосередньо у вікні інтерактивного інтерпретатора irb.

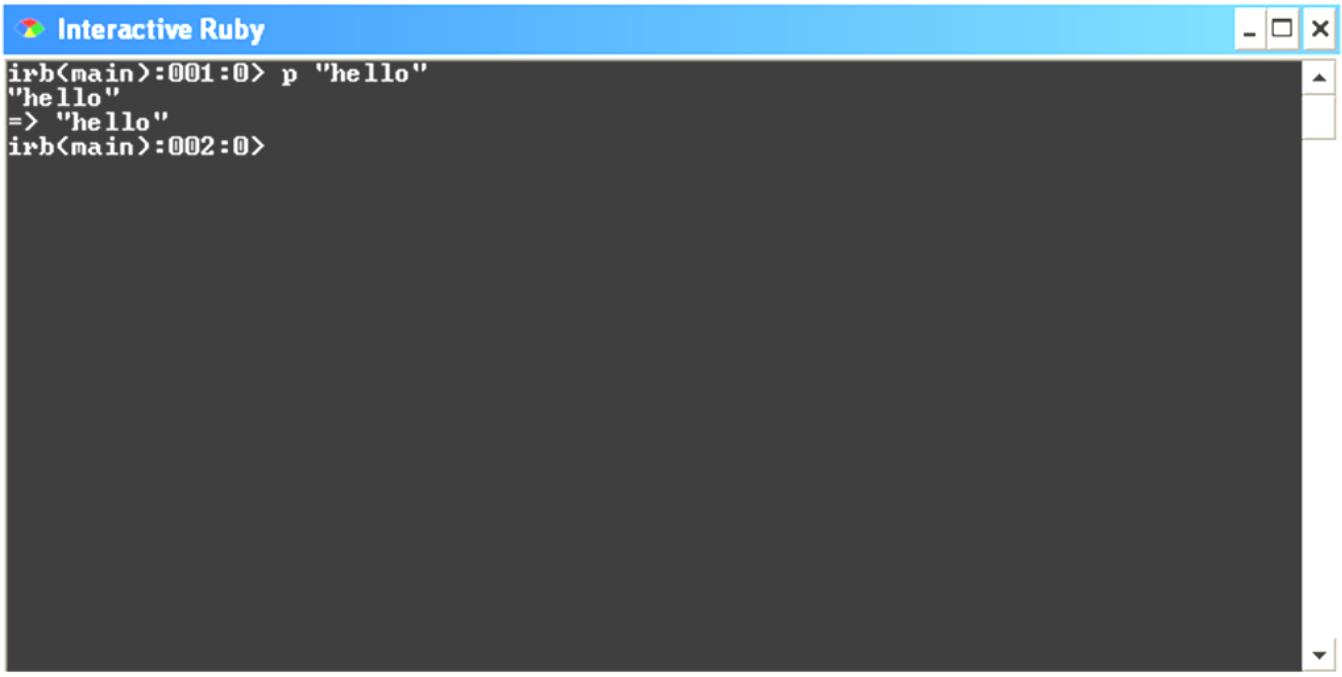


Рис. 1.4. Відображення текстового рядка після виклику оператором «p» у вікні інтерактивного інтерпретатора irb

1.2. Підготовка та виконання програм на мові Ruby

Як правило, в «Командному рядку Ruby» запускаються файли, що мають розширення (.rb). Наприклад, якщо в додатку «Блокнот» ввести рядок з текстом:

```
puts "Hello, World!"
```

і зберегти файл під ім'ям ruby1.rb, то виклик файлу за іменем ruby1 в «Командному рядку Ruby» дасть наступний результат:

```
D:\6w>ruby1
Hello, World!
```

Таким чином, в файлі ruby1.rb міститься програма на мові Ruby. Дана програма повинна мати:

1) ім'я (наприклад, ruby1) і

2) розширення (.rb).

Для виконання (виклику) програми необхідно перенаправити «Командний рядок Ruby» в каталог, обраний для роботи з програмами, написаними на мові Ruby, (наприклад, в каталог 6w на диску Z:).

Для цього використовуються такі команди MS-DOS:

1) команда Z: здійснює переведення інтерпретатора на логічний диск Z:

```
Z:
```

2) команда cd (change directory) здійснює переведення «Командного рядка Ruby» в необхідний каталог (тут – каталог з ім’ям 6w):

```
cd 6w
```

потім в командному рядку вводиться ім’я файлу з програмою на мові Ruby:

```
ruby1
```

Іншим способом виклику програми на мові Ruby в ОС Windows є виконання подвійного клацання лівою клавішею миші по імені програми (файлу) в каталозі, де вона міститься. Щоб вікно з результатами роботи програми не зникало останнім оператором програми повинен бути оператор STDIN.getc, що є оператором очікування кінця введення. У цьому випадку програма складається з двох операторів:

```
puts "Hello, World!"
STDIN.getc
```

Зверніть увагу, що оператор STDIN.getc в кінці програми аналогічний оператору Readln без параметрів в мові Object Pascal і «затримує» вікно «Командного рядка Ruby» на екрані комп'ютера після виконання програми і виведення результату на екран операторами puts, p і ін.

При наявності даного оператора, після виконання програми необхідно одноразово натиснути клавішу Enter.

1.3. Коментарі в програмах

Коментарем називається частина програмного коду, що пропускається при обробці (інтерпретації або компіляції). В Ruby знаком початку коментаря служить символ # (Шарп). Все, що розташовується між символом # і кінцем рядка пропускається. приклад:

```
puts 2 + 2                          # це коментар
puts "Hello!"                 # це також коментар
```

Багаторядкові коментарі поміщаються між словами = begin і = end
Наприклад:

```
= begin
     Це великий коментар
     Дуже великий
= end
          puts "Hello!"
```

1.4. Виведення даних на екран

У мові Ruby існує багато методів виведення даних на екран:

print, printf, p, puts, display та інші.

Для початку будемо використовувати два з них:

• метод puts. Після виведення рядка даних здійснює переклад курсору на новий рядок. Якщо доводиться виводити об'єкт, який не є рядком, то puts викликає метод .to_s для перетворення його до строчному типу;

• метод p. Має найкоротший назву, через що часто використовується для налагодження програм. Так само, як і puts, виконує переведення курсору на новий рядок після виведення даних. Перед вивідом будь-якого об'єкта (в тому числі і рядка) на екран, викликає метод .inspect.

Нижче наведені приклади виведення на екран результатів виконання операторів мови Ruby.

(**Увага**: дана послідовність операторів (команд) повинна міститися у файлі з розширенням (.rb, наприклад, у файлі з ім'ям ruby3.rb).

Після символів ознаки коментарів наведені результати виконання операторів, що вводяться:

```
puts [1, 2, 3, 4]     #=> 1\n2\n3\n4
                      # \n означаеє перенесення рядка
p [1, 2, 3, 4]        #=> [1, 2, 3, 4]
puts "Hello!"         #=> Hello!
p "Hello!"            #=> "Hello!"
puts 5                #=> 5
p 5                   #=> 5
```

Як видно з прикладу, результати під час виведення рядків і масивів істотно розрізняються. Основною відмінністю операторів puts і p є наступна особливість:

- інструкція puts "Hello!" виводить текст: Hello!
- інструкція print "Hello!" виводить текст: Hello!
- інструкція p "Hello!" виводить текст: "Hello!"

Іншими словами, оператор p, виводить дані, обрамлені лапками.

1.5. Формування імен змінних

У програмах на мові Ruby початковий символ імені змінної істотно впливає на контекст її подальшого використання (див. табл.1.1). Даний префікс повідомляє про область дії змінної, а також дозволяє позбутися від круглих дужок при викликах методів.

Таблиця 1.1.

Вплив першого символу імені змінної на контекст іі використання

Початковий символ (префікс)	**Контекст використання змінної**
$	глобальна змінна(наприклад: $s)
@	змінна екземпляра (поле об’єкта, дані–члени) (наприклад: @s)
@@	змінна класа (наприклад: @@s)
a–z або ‘_’	локальна змінна (наприклад: s)
A–Z	константа (наприклад: S)

Крім того існують ще дві псевдозмінні, які є винятком з вищевказаного правила – це змінні з іменами **self** і **nil**.

– self – являє собою посилання на поточний виконуваний об'єкт;

– nil – а) не має значення, б) має значення FALSE або в) має значення неініціалізованої змінної.

1.6. Базові типи даних

Всі дані будь-якого типу в Ruby є об'єктами тих чи інших класів. Взагалі, в Ruby **всі сутності**, в кінцевому рахунку, належать класу **Object**.

Найбільш використовуваними вбудованими типами даних є:

- Fixnum (цілі числа, менші 2^{30}),
- Bignum (цілі числа, більші 2^{30}),
- Float (числа з плаваючою точкою / коми),
- Array (масиви),
- String (рядки),
- Hash (асоціативні масиви).

Природно, що це тільки базові типи, але їх цілком вистачає для вирішення широкого спектра завдань. У математичних обчисленнях цілком достатньо використання цілих чисел (Fixnum і Bignum) і дійсних чисел (чисел з плаваючою точкою/комою – Float).

1.7. Представлення чисел

Числа в Ruby можуть представлятися таким чином:

```
   5          # ціле число
 -12          # негативне ціле число
  4.5         # число з плаваючою комою (дійсне число)
3.092e+5      # число з плаваючою точкою, представлене в логарифмічній (експоненційній) формі.
```

Всі операції в Ruby реалізовані у вигляді **методів**. Основними операціями для дійсних типів чисел є (в порядку збільшення пріоритету): + (додавання) і - (віднімання); * (множення) і / (ділення); ** (зведення в ступінь, піднесення до степеня).

Для цілих типів чисел операції такі: + (додавання) і – (віднімання); * (множення), / (цілочисельне ділення) і % (знаходження залишку від цілочисельного ділення); ** (зведення в ступінь, піднесення до степеня).

Наприклад:

```
11 / 2    # =>  5
11 % 3    # =>  2
```

Важливо пам'ятати, що результат взаємодії цілого і дійсного чисел завжди є дійсне число! На рисунку 1.5 представлено обчислення суми цілого і дійсного (дрібного) чисел.

```
D:\6w>puts 2+3.5
"puts" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.

D:\6w>irb
irb(main):001:0> puts 2+3.5
5.5
=> nil
irb(main):002:0>
```

Рис. 1.5. Обчислення суми чисел: цілого (2) і дійсного (3.5)

У даному сценарії представлено відміну принципів роботи двох інтерпретаторів Ruby. У інтерпретаторі «Командний рядок з Ruby» вводиться оператор «puts 2 + 3.5». Вищевказаний інтерпретатор діагностує помилку. Але після виклику на ім'я «irb» в середині «Командного рядка з Ruby» викликається «Інтерактивний інтерпретатор Ruby» (irb). Даний інтерпретатор виводить результат виконання введеного оператора «puts 2 + 3.5».

1.8. Перетворення рядка тексту в число

Є два основних способи перетворити рядок у число:

– застосувати методи Integer або Float модуля Kernel
– застосувати методи to_i або to_f класу String.

Слід мати на увазі, що імена, які починаються з великої літери, наприклад Integer, зазвичай резервуються для спеціальних функцій перетворення. Нижче наведено приклад застосування перерахованих вище методів (після символу коментаря представлений результат виконаного перетворення):

```
x = "123".to_i     # 123
y = Integer("123") # 123
x = "3.1416".to_f  # 3.1416
y = Float("2.718") # 2.718
```

Обидва методи Float і to_f можуть застосовуватися до чисел, представлених в логарифмічному або експоненційному (інженерному) форматі:

```
x = Float("6.02e23")   # 6.02e23
y = "2.9979246e5".to_f # 299792.46
```

Методи `to_i` та `Integer`, по-різному перетворюють значення, представлені в різних системах числення. За замовчуванням, природно, мається на увазі уявлення чисел в позиційній системі числення з основою `10`, але за їх допомогою можуть оброблятися дані, представлені в деяких інших системах числення (це справедливо і для чисел з плаваючою точкою). Нижче наведено приклад програми, що реалізує алгоритм обчислення об’єму циліндра і демонструє використання методу `.to_i` і глобальні змінні (імена яких починаються з символу $). Важливо знати, що метод `chomp` (він використовується також в мові `Perl`) прибирає з рядка останній символ, який вказує на дію переведення на новий рядок (`\n`), який автоматично додається командою `gets` при введенні. Метод використовується також в тих випадках, коли потрібно перетворити введений рядок в число, тобто в «голі цифри».

У рядку «`$sr = gets.chomp`» створюється глобальна змінна `$sr`, далі командою `gets` вводиться рядок символів з екрану комп'ютера. При цьому у нього відсікається символ переведення рядка за допомогою `chomp` і методом `to_i` її вміст перетвориться в числове значення, яке потім присвоюється змінної `$sr`.

```
#Lab_1

print “Enter radius:/n”  #нагадування ввести радіуса круга.
$sr = gets.chomp         #занесення значення до змінної $sr
print “Enter height:/n”  #нагадування ввести висоту цидіндра
$sh = gets.chomp         # занесення значення до змінної $sh
$r = $sr.to_s.to_i       #конвертація тексту до цілого числа
$h = $sh.to_s.to_i       #конвертація тексту до цілого числа
Puts “Numbers =”, $r, $h
$v = 3.14*$r* $r*$h      #обчислення об’єму циліндра
Print “v=”+$v.to_s, “/n” #виведення результату
```

Результат роботи програми наведено на наступному рисунку 1.6.

```
D:\6w>lab1
Enter radius:
4
Enter height:
7
Numbers=
4
7
v=351.68
```

Рис. 1.6. Результат роботи програми

Досить нетривіально в мові `Ruby` відбувається отримання модуля заданого числа, з урахуванням того, що «... в мові `Ruby` все представляється об'єктами і обробляється відповідними методами». Нижче наведено код прикладу використання методу `abs()` для отримання модуля числа.

```
q = Math.sqrt(0.3)
      r = - Math.cos(0.3)
  r = (r).abs
puts "Sqr, Cosinus=", q, r
```

```
D:\6w>cosabs
Sqr, Cosinus=
0.5477225575051661
0.955336489125606
```

Рис.1.7. Знаходження абсолютної величини змінної `r`

Застосування стандартних математичних функцій в мові `Ruby` описано далі.

1.9. Використання математичних функцій

Для обчислення значень стандартних математичних функцій (які завжди є методами) в мові `Ruby` використовується модуль `Math`. Це вбудований модуль для математичних обчислень. Взагалі, модулі грають дві різні ролі в мові `Ruby`. Тут показана одна з ролей: угруповання схожих методів разом під загальною назвою. Модуль `Math` містить також і такі методи, як `sin()`, `tan()` і багато інших. Звернення до відповідних методів виконується шляхом використання точки, яка відділяє ім'я відповідної функції від імені модуля `Math`. Наприклад – `Math.sqrt(9)`. У цьому виразі точка ідентифікує одержувача повідомлення. В даному випадку повідомлення, це вираз `sqrt(9)`, який означає виклик методу `sqrt` (скорочення від «`square root`» – квадратний корінь) з параметром `9`.

Результатом виклику цього методу буде значення `3.0`. Результат не дорівнює цілому числу `3`. Даний факт означає, що в більшості випадків квадратний корінь числа не є цілим числом. Таким чином, цей метод завжди повертає число з плаваючою точкою (дійсне число). Приклади звернень до деяких інших стандартних функцій наведено в наступному фрагменті програми.

```
q = Math.sqrt(0.3)
r = Math.cos(0.3)
puts "Sqr, Cosinus=", q, r
c = Math.tan(0.3)
d = Math.atan(0.3)
puts "Tan, Atan=", c, d
q = Math.exp(1.0)
r = Math.log(2.71)
puts "Exp, Log naturalny =", q, r
```

```
Sqr, Cosinus=
0.5477225575051661
0.955336489125606
Tan, Atan=
0.3093362496096232 5
0.2914567944778671
Exp, Log naturalny=
2.718281828459045
0.9969486348916096
Press ENTER to close the window...
```

Рис.1.8. Результат роботи програми

1.10. Оператори Ruby.

У таблиці 1.2. наведено список операторів мови Ruby, які представлені в порядку зменшення пріоритету.

Таблиця 1.2.

Оператори мови Ruby, розташовані у порядку зменшення пріоритету

Оператор	Властивості оператора
`::`	Дозвіл області видимості
`[]`	Взяття індексу
`**`	Зведення в ступінь
`+ - ! ~`	Унарний плюс / мінус, НЕ ...
`* / %`	Множення, ділення ...
`+ -`	Додавання / віднімання
`<< >>`	Логічні зсуви ...
`&`	Порозрядне І
`| ^`	Порозрядне АБО, виключаюче АБО
`> >= < <=`	Порівняння
`== === <=> != =~ !~`	Рівність, нерівність ...
`&&`	Логічне І
`||`	Логічне АБО
`.. ...`	Оператори діапазону
`= (also +=, -=, …)`	Привласнення
`?:`	Тернарний вибір
`not`	Логічне заперечення
`and or`	Логічне І, АБО

1.11. Завдання та вміст звіту до даної лабораторної роботи.

Завдання: Студент повинен виконати програмування математичної функції згідно свого варіанту. Варіант вибирається з таблиці завдань у додатку до лабораторних робіт та відповідає номеру комп'ютера, за яким працює студент.

Зміст звіту до лабораторної роботи:

1. Назва роботи.
2. Тема, мета і завдання до роботи.
3. Вихідний код розроблених та збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

ЛАБОРАТОРНА РОБОТА №2.

Особливості роботи з чисельними типами даних.

Тема: Рішення математичних задач з використанням цілих і дійсних чисел.

Мета: Використання інтерпретаторів Ruby для налагодження програм. Обробка цілих і дійсних типів даних і застосування основних методів та математичних функцій. Введення і виведення даних з використанням екрану.

Теоретичні відомості.

2.1. Використання базових типів даних

У мові Ruby всі сутностіі є об'єктами деяких класів. Тому будь-яка операція з обробки будь-якого компонента даних в програмі є методом.

Іноді, щоб уточнити необхідність або правомірність застосування того чи іншого методу існує можливість дізнатися яким класом породжений обробляється об'єкт. Для цієї мети служить метод `.class`, який повертає ім'я класу, якому належить даний об'єкт. наприклад:

```
"типовий рядок".class    #-> String
0xface.class             #-> Fixnum
9876543210.class         #-> Bignum
Fixnum.class             #-> Class
```

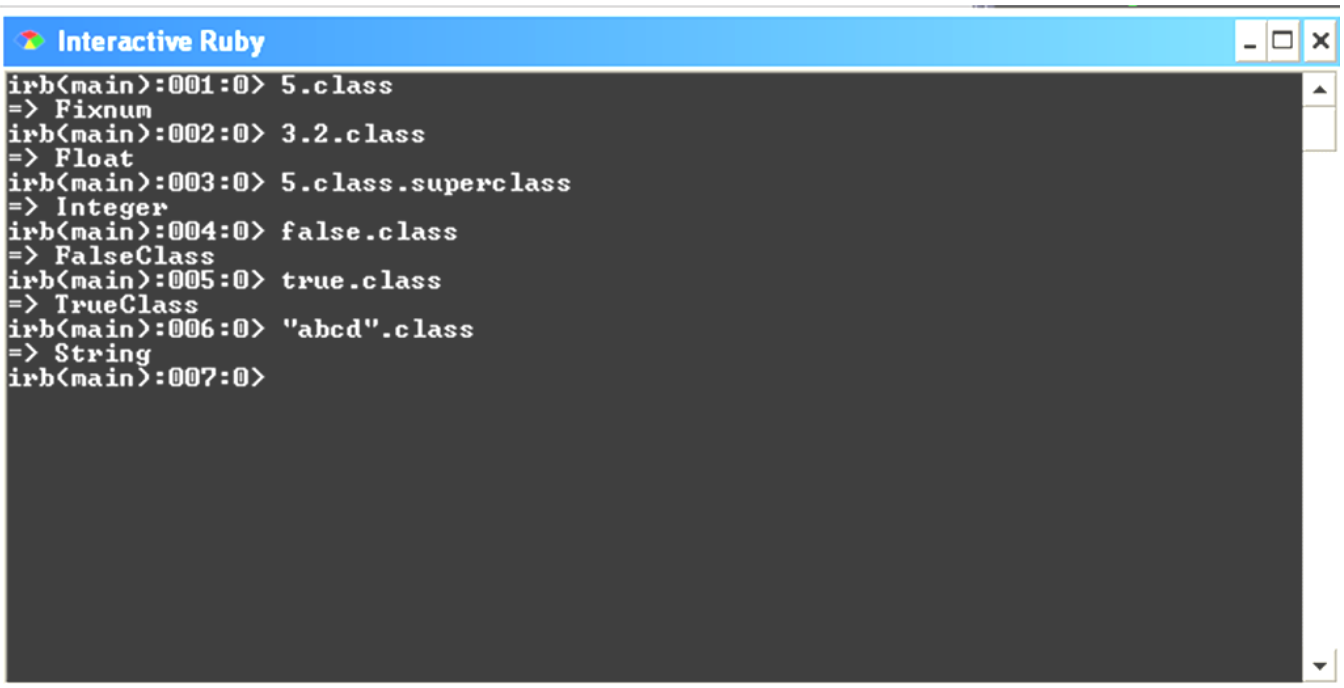


Рис. 2.1. Результат звернення до методу `.class` в інтерпретаторі мови Ruby

На рис.2.1 можна побачити, що іншим інструментом дослідження класової ієрархії є метод `.superclass`, який призначений для отримання імені базового класу (класу батька).

Важливо знати, що метод `.superclass` можна викликати тільки для об'єкта класу `class`, який, в свою чергу, можна отримати за допомогою самого методу `.class`.

Виходячи з вищесказаного будь-яка операція з обробки даних (наприклад, +, -, *, / і.т.д.) є методом (тобто викликається для деякого об'єкта) і завжди повертає значення. В остаточному підсумку, всі сутності в мові `Ruby`, належать класу `Object` (рис. 2.2).

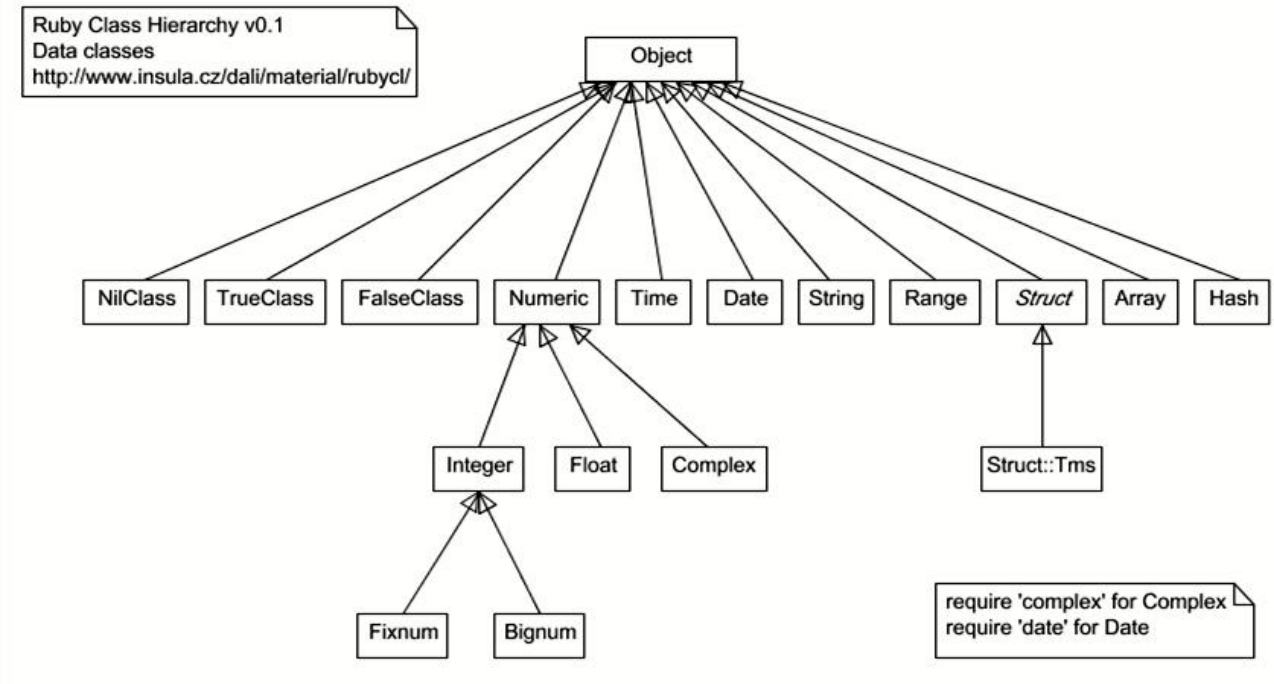


Рис.2.2. Назви основних класів, які породжують типи даних, що використовуються в мові `Ruby`.

2.2. Об'єкти і методи в Ruby

Як і в інших об'єктно-орієнтованих мовах, всі елементи оброблюваних даних є об'єктами. При цьому кожен об'єкт є представником (екземпляром) деякого класу і його поведінка (функціональність) визначається саме їм.

До простих об'єктів мови `Ruby` можна віднести:

– `true`- логічна величина, що означає «`істина`»; єдиний представник класу `TrueClass`;

– `false`- логічна величина, що означає «`неправда`»; єдиний представник класу `FalseClass`;

– `числа`- представники класу `Numeric`, який в якості підкласів містить дробові (дійсні) (`Float`) і цілі числа (`Integer`);

– `рядки` - представники класу `String`;

– `nil`– «ніщо»; єдиний представник класу `NilClass`.

Незважаючи на візуальну простоту створюваних програмних кодів, інфраструктура програм, які запускаються на виконання в `Ruby` є досить розвиненою. В `Ruby`-співтоваристві, сутності, взаємодіючі при запуску програми, називаються «`Всесвіт` (`Space`)».

Під час роботи програми «Всесвіт» розвивається завдяки взаємодії між її окремими компонентами (об'єктами). Об'єкти взаємодіють один з одним, обмінюючись повідомленнями, або, як ще кажуть, виконуючи методи.

Для застосування методу до числовому об'єкту використовується оператор виклику методу, що позначається символом «.» (крапка). Таким чином, після зазначення імені або «образу» об'єкта, далі розміщується точка, а потім вказується найменування (ім'я) застосовуваного методу (наприклад,

(-2).abs). В даному випадку до негативного числа (об'єкту) «-2» застосовується метод .abs (тобто знаходиться модуль числа -2).

У разі застосування простих арифметичних операцій (додавання, множення і т.д.) синтаксис їх застосування може враховувати їх об'єктно-орієнтовану суть. Ось найпростіші приклади взаємодії об'єктів і методів:

```
 2.+(3)    => 5
"a".*(5)   => "aaaaa"
```

У першому випадку числу (об'єкту 2) було послано повідомлення (або викликаний метод - ".+3"). В результаті був створений новий об'єкт - число 5.

У другому випадку для рядка (об'єкта "a") був викликаний метод .*(5), що означає помножити на 5.

2.3. Простір об'єктів

Отже, Ruby-програма, при запуску на виконання, створює свій власний Всесвіт. Дізнатися, що ж міститься в просторі оточення простої Ruby-програми, можуть допомогти методи модуля ObjectSpace, які, в тому числі, повертають загальну кількість об'єктів в запущеній (виконуваній) програмі. При цьому методи модуля ObjectSpace ніколи не показують так звані Immediate objects – об'єкти true, false, nil і всі цілі числа, які є об'єктами класу Fixnum.

Нижче наведено текст програми, у якій змінної x присвоюється значення 77.777. Звернення до методу .count_objects модуля ObjectSpace, показує, як багато різних об'єктів виникає після її старту. Нижче наведені текст програми та результат її роботи.

Програма:

```
x = 77.777
puts ObjectSpace.count_objects
```

Результат виконання програми:

```
{:TOTAL=>14717, :FREE=>497, :T_OBJECT=>8, :T_CLASS=>542,
:T_MODULE=>21, :T_FLOAT=>8, :T_STRING=>6263, :T_REGEXP=>24,
:T_ARRAY=>982, :T_HASH=>14, :T_BIGNUM=>3, :T_FILE=>9,
:T_DATA=>397, :T_MATCH=>106, :T_COMPLEX=>1, :T_NODE=>5823,
:T_ICLASS=>19}
```

Всього об'єктів в Ruby-всесвіту для цієї програми, виявляється 14717. Серед них 8 дійсних чисел (Float), 6263 рядків (String), 982 масива (Array), 3 об'єкти класу Bignum, 9 об'єктів типу File і т.д.

Для того, щоб вивести, наприклад, тільки об'єкти класу Float слід трохи змінити програму.

Програма:

```
# encoding: UTF-8
x = 77.777
count = ObjectSpace.each_object(Float){|obj| p obj}
puts "Всего объектов класса Float: #{count}"
```

Результат виконання програми:

```
77.777
2.718281828459045
3.141592653589793
NaN
Infinity
2.220446049250313e-16
1.7976931348623157e+308
2.2250738585072014e-308
Всего объектов класса Float: 8
```

Серед цих восьми об'єктів знаходиться і те число, яке є в тексті програми: 77.777.

2.4. Типи числових даних, форма запису чисел, присвоювання значень і зміст змінних (об'єктів)

Як відомо, в більшості мов програмування тип даних визначає:

– спосіб представлення елементів сукупності (елемент: цифровий, текстовий, логічний та ін.);

– безліч допустимих значень змінних даного типу (тобто кордону зміни). Наприклад, в мові Delphi цілі значення типу Shortint змінюються в межах між -128 та +128;

– безліч допустимих операцій, які застосовуються до даного типу даних (наприклад, складання, цілочисельне ділення, логічне множення, операція конкатенації і ін.);

– розмір області пам'яті, яку займаэ змінна даного типу. Наприклад, в мові Delphi є сім (!) цілих типів даних: Shortint, Smallint, Longint, Int64, Byte, Word і Longword. Мінімальний розмір області пам'яті відводиться для типу Shortint – один байт, а максимальний – для типу Int64 – 8 байт.
В Ruby, на відміну від багатьох інших мов програмування, і, в тому числі, мови Delphi, немає жорсткої типізації.

Це означає, що тип змінної визначається динамічно і може змінюватися в процесі виконання програми. Якщо в Delphi змінна цілого типу приймає і забезпечує обробку значень даних цілого типу протягом усього життя (виконання) програми, то в мові Ruby тип і зміст змінної визначаються динамічно і можуть змінюватися по ходу програми.

У таблиці наведені фрагменти програм на мовах Delphi і Ruby, які демонструють зміни змісту змінної з ідентифікатором **a**.

Таблиця 2.1.

Порівняння можливостей використання змінних у мовах програмування Delphi і Ruby

Delphi	**Ruby**

var
a : Integer;
Begin
a := 6.0 * 4;
a := 18 div a;
a := 45.762 div a;
Writeln ('a = ', a);
End.
	a = 2
	print "a= ", a, "\n"
	a ="abc"
	print "a= ", a, "\n"
	a = 32.89e-16
	print "a= ", a, "\n"

Примітка: Протягом всього часу виконання програми у змінній a містяться тільки цілі значення	**Примітка**: Протягом роботи інтерпретатора змінна a приймає різні значення (цілі, речові, рядкові)

Виконання Ruby-програми з попередньої таблиці дає наступний результат:

```
D:\6w>type1
a= 2
a= abc
a= 3.289e-15
```

2.5. Представлення числових даних в мові Ruby

В Ruby є 5 вбудованих класів і 3 додаткових для подання чисел різних типів і, відповідно різної розрядності.

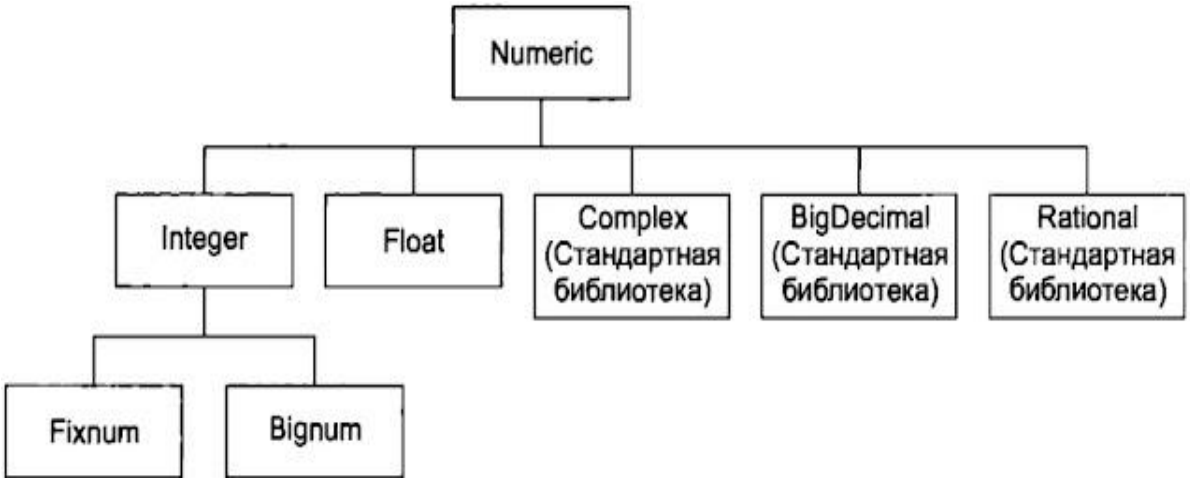


Рис. 2.3. Блок-схема вбудованих класів для подання чисел в мові Ruby

Всі числові об'єкти в Ruby є екземплярами класу Numeric. Всі цілі значення є екземплярами класу Integer. Всі вони, в свою чергу, відносяться або до класу Fixnum (описує цілі числа, чиє двійкове подання здатне розміститися в машинному слові; на більшості комп'ютерів цей розмір дорівнює значенню 31 біт), або є екземплярами класу Bignum (цілі числа довільного розміру). При виконанні різних дій (операцій) над числами і виходом їх розміру за рамки Fixnum, число перетворюється у Bignum. Зазвичай інтерпретатор Ruby автоматично розпізнає для цілих чисел, який тип даних використовувати для заданої змінної і автоматично призводить тип FixNum до Bignum. Щоб перевірити це, виконаємо такий код:

```
type1=1000000
type2 = type1 ** 3
puts type1          # виведення на екран 1000000
puts type2 # на екран виводиться 1000000000000000000
puts type1.class # на екран виводиться Fixnum
puts type2.class # на екран виводиться Bignum
```

Дійсні числа відповідають класу Float, який служить для представлення чисел з плаваючою точкою. Класи Complex, BigDecimal і Rational не вбудовані в Ruby, але поширюються разом з ним як частина стандартної бібліотеки. Для їх використання необхідно підключити відповідну бібліотеку оператором require:

```
require 'bigdecimal' # підключаємо бібліотеку bigdecimal
```

Клас Complex представляє комплексні числа, клас BigDecimal – дійсні числа з довільною точністю. При цьому використовується не двійкове, а десяткове представлення. Клас Rational представляє раціональні числа, які є результатом ділення одного цілого числа на інше. Як випливає з вищесказаного, клас Float вносить істотно більшу похибку в уявлення чисел, ніж класи BigDecimal і Rational. Для перевірки можливостей вищевказаних класів порівняємо результати обчислення звичайного (з точки зору людини) виразу: 3.2-2.0=1.2.

Код програми:

```
a = Float ("3.2")
b = Float ("2.0")
c = Float ("1.2")
puts " Float: a-b=", a-b
   if (a-b) == c then
                    puts "All is well"
else
                    puts "Numbers not equal"
end

require 'bigdecimal'    # підключення бібліотеки bigdecimal
a = BigDecimal ("3.2") ;
b = BigDecimal ("2.0")
c = BigDecimal ("1.2")
puts " BigDecimal: a-b=", a-b
```

```
if (a-b)== c then
                    puts "All is well"
else
                    puts "Numbers not equal"
end
a = Rational ("3.2")
b = Rational ("2.0")
c= Rational ("1.2")
puts " Rational: a-b=", a-b
if (a-b)==c then
                    puts "All is well"
else
                    puts "Numbers not equal"
end
```

Результат виконання коду, приведеного вище:

```
Float: a-b=
1.2000000000000002
Numbers not equal

BigDecimal: a-b=
0.12E1
All is well

Rational: a-b=
6/5
All is well
```

Виявляється, що округлення чисел при перетворенні їх в клас `Float`, а також результатів виконання арифметичних операцій (методів) в цьому класі більш «руйнівний», ніж в класах `BigDecimal` і `Rational`.

Аналізуючи результати виконання вищенаведеного коду можна зробити висновок, що:

– перетворення даних в класі `Float` призводять до досить відчутних погрішностей;

– перетворення даних в класах `BigDecimal` і `Rational` дозволяють досягти дуже високої точності в обчисленнях;

– екземплярами класу `Rational` повинні бути тільки раціональні числа (частка від ділення двох цілих чисел). З ірраціональними числами його використовувати не рекомендується.

2.6. Правила представлення числових даних в мові Ruby

Треба пам’ятати, що при запису (формуванні) різних числових об'єктів у мові `Ruby` використовуються наступні правила.

- Класс `Integer` – це проста послідовність цифр. Наприклад: `0123456789`.
- Об'єктом класу `Fixnum` є цілочисельне значення `N`, яке не може перевищувати значення 2^{30} ($N < 2^{30}$). При виході за цю межу число перетворюється в об'єкт `Bignum`, який підтримує цілочисельні значення будь-якого розміру.

• В цілочисельних літералах[1] допустимо використовувати знаки підкреслення "_" (але тільки не на початку і не в кінці числових послідовностей). Цю властивість можна використовувати для поділу чисел на класи тисяч. Наприклад: "1_000_000_000".

• Якщо число починається з цифри нуль і містить символ «x» – 0x або 0X – воно розглядається як шістнадцяткове. Наприклад: 0xdeadbeef.

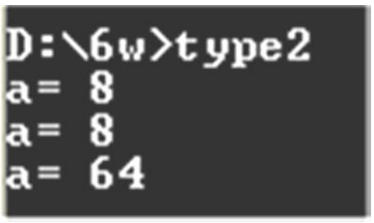
• Якщо число починається з цифри нуль і містить символ «b» – 0b або 0B – воно розглядається як двійкове і може включати в себе тільки цифри 0 і 1. Наприклад: 0b100 (# => 410); а 0b1211 – неправильний запис, тому що цифра 2 не входить в алфавіт двійкової системи числення!

• Якщо число починається з цифри нуль і символу «o» – 0o або 0O (тобто в другій позиції розташовується буква «o») – дане число розглядається як вісімкове. Окрім того, якщо число починається з нуля (0) і не містить в другій позиції букв o (O), x (X) або b (B) – воно є восьмеричним і має складатися тільки з цифр, які змінюються у діапазоні значень від 0 до 7. Наприклад, числові послідовності 01234 і 0o1234 представляють один і той самий об'єкт – 668_{10}.

Приклад привласнення об'єкта – вісімкового числа змінній (об'єкту) ***a*** в програмі і вивід результату присвоювання на екран наведено нижче:

```
a = 010
print  "a= ", a, "\n"
    a =0o10
     print "a= ", a, "\n"
        a = 0o100
        print "a= ", a, "\n"
```

Результат виконнання коду, приведеного вище дає наступний результат:



2.7. Створення даних заданих типів і перетворення чисел

Для завдання необхідних для обробки даних значень змінних і перетворення чисел (числових об'єктів) з одного виду уявлення в інший існує ряд методів. До них відносяться наступні.

.to_f (Float) – метод .to_f використовується для перетворення цілого числа в число з плаваючою точкою/комою. Якщо говорити більш коректно, то цілочислене уявлення замінюється на представлення числа з плаваючою точкою, наприклад:

[1] Літерали є константами, які включаються безпосередньо в текст програми. Наприклад, у операторі a=100, a – змінна, а 100 – це літерал.

```
s = 2.to_f #=> 2.0
t = 2 / 3 #=> 0
r = 2 / 3.to_f #=> 0.6666666666666666
```

Такий самий результат дає застосування метода `Float`:

```
s = Float(2)      #=> 2.0
t = Float("2")    #=> 2.0 (перетворення рядок в число)
r = Float(2/3.0) #=> 0.6666666666666666
```

`.to_f` – перетворює текстовий рядок в число з плаваючою точкою, наприклад:

```
"20 dollars per hour".to_f #=> 20.0
```

`.to_s` – метод, який перетворює число в рядок тексту, наприклад:

```
S = 125.to_s #=> "125"
```

`.to_s (x)` – метод, який перетворює дані в рядок тектсу, але дозволяє задавати необхідну систему обчислення у вигляді параметра `x` (тільки для цілих чисел), наприклад:

```
# пертворення числа в двійкове представлення
1.to_s(2) #=> "1"
2.to_s(2) #=> "10"
889.to_s(2) #=> "1101111001"
# пертворення числа в шістнадцядкове представлення
8.to_s(16) #=> "8"
9.to_s(16) #=> "9"
23.to_s(16) #=> "17"
998.to_s(16) #=> "3e6"
# пертворення числа в вісімкове подання
8.to_s(8) #=> "10"
12.to_s(8) #=> "14"
# перетворення числа у будь-яке потрібне подання
1.to_s(12) #=> "1"
13.to_s(12) #=> "11"
```

`.to_i` і `.to_int (Integer)` перетворюють об'єкт до цілочисельного типу, наприклад:

```
1.to_i #=> 1
1.1.to_i #=> 1
5.9.to_i #=> 5
5.9.to_int #=> 5
"20 dollars per hour".to_i #=> 20
s = Integer (79.43) # 79
t = Integer (9.25) # 9
```

При підключенні бібліотеки `bigdecimal` можливо перетворення чисел в типи `BigDecimal` і `Rational`. Наприклад, можливі такі перетворення:

Таблиця 2.2.

Приклади перетворень типів даних

Початковий тип	Метод	Тип, в який перетворюється початковий
`String`	`BigDecimal()`	`BigDecimal`
`String`	`Rational() / .to_r`	`Rational`
`BigDecimal`	`Rational() / .to_r`	`Rational`
`Rational`	`BigDecimal()`	`BigDecimal`

2.8. Арифметичні операції з цілими і дробовими числами

Всі арифметичні операції в мові `Ruby` реалізовані у вигляді методів, однак, незважаючи на це, кожен оператор має свій пріоритет виконання. Наприклад, як і в математиці, у метода, що виконує множення – пріоритет більше, ніж у методу, який виконує додавання. Таким чином, +, -, *, /, ** – це методи знаходження суми, різниці, добутку, результату ділення (частки) і піднесення до степеня.

Одним з найважливіших моментів у взаємодії числових об'єктів є об'єднання в обчисленнях дробових (дійсних) і цілих чисел. Якщо хоча б одне з чисел в обчисленнях є дійсним, результат буде теж дійсним, наприклад:

```
6   + 4      #=> 10
6.0 + 4      #=> 10.0
6 - 4.0      #=> 2.0
6.0 * 4.0    #=> 24.0
```

Іншою особливістю обчислень в мові `Ruby` є різноманіття в використанні методів (операторів) ділення. В `Ruby`-програмах можуть застосовуватися три різні способи ділення (`/`, `.div` і `.fdiv`).

При використанні першого з них (`/`) слід мати на увазі наступне. Якщо ділене і дільник – цілі числа, то результат буде інтерпретуватися як результат цілочисельного ділення, наприклад:

```
1/3    #=> 0
2/3    #=> 0
3/3    #=> 1
5/2    #=> 2
```

Якщо хоча б одне з чисел є дійсним, то результат ділення також є дійсним:

```
1.0 / 3    #=> 0.3333333333333333
2 / 3.0    #=> 0.6666666666666666
3.0 / 3    #=> 1.0
5.0 / 2    #=> 2.5
10 / 3.5 #=> 2.857142857142857
```

Метод `.div` завжди повертає результат цілочисельного ділення, тобто без дробової частини.

```
5.div 3.0    #=> 1
```

Метод `.fdiv` завжди повертає дійсний результат від ділення довільних поєднань цілих і дробових чисел (ціле на ціле, ціле на дробове і дробове на ціле). наприклад:

```
5.fdiv 3.0 #=> 1.6666666666666667
```

2.9. Отримання випадкових чисел

При вирішенні інженерних задач часто виникає задача отримання випадкових чисел в різних діапазонах їх зміни. наприклад:

```
rand(100)     #=> 86
rand          #=> 0.599794231588021
```

У першому випадку метод `rand` повертає ціле число в діапазоні від `0` до `99` (на одиницю менше `100`). У другому випадку метод `rand` повертає число з плаваючою комою в діапазоні від `0.0` до `1.0` включно. Різниця в результаті обумовлено переданим параметром:

• якщо передається параметр (в даному випадку `100`), то генерується ціле випадкове число (в діапазоні `0 ... N-1`, де `N` – аргумент, що передається);

• якщо параметр відсутній, то генерується число з плаваючою комою в діапазоні від `0.0` до `1.0`.

Є спосіб передбачити весь ряд «випадкових» чисел. Робиться це за допомогою методу `srand`. Йому передається ціле число (ідентифікатор «випадкової» послідовності). Після цього весь випадковий ряд можна передбачити:

```
srand 123
Array.new(5){ rand(100) }    #=> [66, 92, 98, 17, 83]
```

2.10 Не математичні функції для перетворення даних

Для того, щоб поміняти місцями цілочисельні значення двох змінних без використання третьої змінної можна використати наступний підхід:

```
number_one, number_two = 134, 234
number_one, number_two = number_two, number_one
```

Поміняти порядок цифр даного числа на зворотний:

```
start_number = 1234
puts start_number.to_s.reverse   # метод reverse перевертає рядок
```

Дано ціле число. Необхідно визначити, чи є воно парним:

```
print("Vvedite tseloe chislo: ")
c = gets.to_f.round.to_i
if (c % 2 == 1)
        print("Chislo #{c} yavleaetsea nechiotnym")
else
        print("Chislo #{c} yavleaetsea chiotnym")
end
STDIN.getc
```

Слід пам'ятати, що імена Ruby-методів можуть закінчуватися знаком питання та знаком оклику. Знак питання використовується для позначення предикатів – методів, які повертають булеві значення (true або false). Наприклад, в класах Array і Hash визначені методи за іменем empty?, які перевіряють наявність в структурі даних будь-яких елементів.

Знак оклику в закінченні імені методу є ознакою того, що при використанні методу слід дотримуватися особливої обережності. У деяких базових класах Ruby визначені пари методів з однаковими іменами з тією лише різницею, що у одного імені в закінченні розміщено знак оклику, а в іншого – ні. Звичайний метод, який не має знаку оклику, повертає модифіковану копію об'єкта, для якого він викликаний, а метод зі знаком оклику є методом-мутатором, який змінює безпосередньо сам об'єкт

2.11. Завдання до лабораторної роботи

Завдання для програмування береться з файлу додатку до Лабораторних робіт. Номер варіанта відповідає номеру комп'ютера, на якому працює студент. Звіт до даної лабораторної роботи повинен містити.

1. Назву роботи.
2. Тему, мету і завдання до роботи.
3. Вихідний код збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

ЛАБОРАТОРНА РОБОТА №3.

Керуючі структури мови Ruby
(Умовні оператори, цикли і методи-ітератори).

Тема: Використання керуючих операторів в рішенні логічних і математичних задач.

Мета: Використання інтерпретаторів `Ruby` для написання і налагодження програм.

Теоретичні відомості.

3.1. Отримання і використання логічних значень.

Як відомо, логічний (булевский) тип – це тип даних, які мають тільки два значення: «`істина`» або «`неправда`». Через особливості організації та взаємодії об'єктно-орієнтованих сутностей в мові `Ruby`, звичайний для усіх інших мов програмування тип логічних змінних `Boolean` – тут відсутній. Однак, оскільки булеві оператори можуть застосовуватися до будь-яких значень об'єктів мови і повинні мати незмінну поведінку для будь-якого типу операндів-об'єктів, в `Ruby` визначені спеціальні значення `true` і `false`. Для забезпечення роботи булевих операторів з даними спеціальними значеннями `true` і `false` прийнято угоду, що:

- `false` і `nil` вважаються помилковими значеннями (`false`);
- всі інші значення, включаючи `true`, `0`, `""`, `[]` і `{}` – вважаються істинними (`true`).

Значення логічного типу найчастіше з'являються в результаті виконання відповідних методів (операцій) порівняння: *дорівнює*, *не дорівнює*, *більше*, *менше*, *більше або дорівнює*, *менше або дорівнює*).

Таблиця 3.1.

Методи порівняння в мові Ruby:

Назва методу	Символи позначення
дорівнює	= =
не дорівнює	!=
меньше	<
більше	>
меньше чи дорівнює	<=
більше чи дорівнює	>=

У мові `Ruby`, традиційно імена логічних методів закінчуються символом «?» (знак питання). Наприклад, метод `.eql?`.

Для формування складних логічних умов (виразів, висловів) застосовуються так звані логічні зв'язки або оператори (операції). Логічними операторами (`not`, `and`, `or` і `xor`) називаються оператори, які можуть виконуватися з логічними константами, змінними, виразами і значеннями (`true` або `false`). Результати виконання логічних операторів з двома операндами `a` і `b`, що мають значення `true` і `false` в різних поєднаннях, наведені в таблиці нижче.

Таблиця 3.2.

Таблиця істиності. Результати виконання операцій `not`, `and`, `or` та `xor`

Логічні аргументи, які приймають значення true та false		Результат виконання відповідних логічних операцій з різними комбінаціями значень аргументів a і b			
a	**b**	**not a**	**a and b**	**a or b**	**a xor b**
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

При виконанні логічних операцій найвищий пріоритет має унарна операція заперечення (`NOT`). Потім йде операція логічного множення (`AND`), а найнижчий пріоритет мають операції `OR` і `XOR`. При обчисленні складних логічних виразів, загальна послідовність виконання різних операцій, з урахуванням їх пріоритету, прийнята такою:

- першими виконуються арифметичні операції (**, *, /, +, -) і, можливо, деякі інші;
- далі виконуються методи порівняння (= =, ! =, >, <, >=, <=), які повертають логічні значення `true` і `false`;
- і потім виконуються логічні операції (їх назви та пріоритети виконання наведені далі в таблиці 3.3).

При виконанні дій з логічними методами і логічними значеннями в мові `Ruby` слід мати на увазі кілька важливих особливостей взаємодії операцій, пов'язаних з пріоритетом їх виконання. Логічні операції (методи) розташовані у таблиці в порядку зниження пріоритету розташовуються в наступному порядку:

Таблиця 3.3.

Відносні пріоритети логічних операцій

Позначення операції	**Назва операції**	**Відносний номер пріоритету**
!	заперечення	1
&&	логічне множення (ЛМ)	2
||	логічне складання (ЛС)	3
=	присвоювання	4
not	заперечення	5
or and	ЛС та ЛМ	6

Такий складний розподіл за рівнями пріоритету призводить до необхідності з особливою ретельністю стежити за використанням логічних операцій при формуванні логічних умов і виразів, так як в мові Ruby істотно різниться застосування оператора присвоювання для відправлення логічних даних в об'єкти-змінні при використанні різних типів логічних операторів.

Таким чином, оскільки для одних і тих же дій використовуються і методи, і операції (заперечення: `!` і `not`; логічне множення: `&&` і `and`; логічне складання: `||` і `or`), а **пріоритети** цих інструментів **різні** – виконання логічних перетворень у мові програмування Ruby значно ускладнене і суттєво відрізняється від інших мов.

Для порівняння розглянемо виконання логічних операцій у мові `Visual Basic for Application` (`VBA`) та порівняємо його з ідентичним кодом на `Ruby`.

```
Public Sub Log1()
Dim a, b, c, s1, s2, s3 As Boolean
a = False: b = True: c = True
s1 = a Or b And c
s2 = a Or (b And c)
s3 = (a Or b) And c

Debug.Print " a, b, c =", a, b, c
Debug.Print " s1 =", s1
Debug.Print " s2 =", s2
Debug.Print " s3 =", s3
```

```
Immediate
 a, b, c =     False          True           True
 s1 =          True
 s2 =          True
 s3 =          True
```

.

Рис. 3.1. Обчислення логічних виразів в мові VBA

```
#=========================================================
a = false ;  b = true ;  c = true
#=========================
s1 =  a or b and c
s2  = a or (b and c)
s3 = (a or b) and c

print " 1=", s1, "\n"
print " 2=", s2 ,"\n"
print " 3=", s3 ,"\n"
#=========================
```

```
a = false ;  b = true ;  c = true
#=========================
s1 =  a || b && c
s2  = a || (b && c)
s3 = (a || b) && c

print " 1=", s1, "\n"
print " 2=", s2 ,"\n"
print " 3=", s3 ,"\n"
#=========================
```

```
D:\6w>log7
 1=false
 2=false
 3=true
```

```
D:\6w>log8
 1=true
 2=true
 3=true
```

Рис. 3.2. Виконання груп функціонально однакових логічних операторів в мові `Ruby`, які дають неправильний результат

У першому фрагменті коду на рис. 3.2., використовуються оператори `or` і `and`, а в другому – оператори `||` і `&&`. Як видно, результати не збігаються. Все це відбувається тому, що в таблиці пріоритетів операторів мови `Ruby` (див. табл. 3.3.), оператори `&&` і `||` стоять вище операторів `and` і `or`, які, до того ж знаходяться на одному рівні пріоритету. Але що більш важливо, оператори `and` і `or` стоять по пріоритету нижче оператора присвоювання `=` .А це провокує в коді програми помилки наступного типу.

Оскільки оператор `=` має більший пріоритет, ніж оператор `or`, `Ruby` спочатку присвоює значення `false` змінный `S1`, а потім порівнює її зі значенням `b` (`true`). Тому виходить так, що замість значення `true` в змінної `S1`, заноситься значення `false`. Такий же код, але з оператором `||` замість `or`, дає правильні результати. Тому рекомендується замінювати оператори `and`, `or` і `not` на оператори `&&`, `||` та `!`.

Слід зазначити ще один непростий момент. Незважаючи на відсутність оператора присвоювання (=) в наведених нижче операторах puts (див. рис. 2.13), оператори and, or і not працюють тут так само неправильно. Навпаки, оператори &&, || і! продовжують давати коректні результати. Але, використання тих же операторів and, or і not в умовних операторах дає правильний результат (рис. 2.14).

```
#=============================================
a= false ; b= true; c= true

puts " 1=",  a || b && c  ; puts " 2=",  a or b and c
puts " 3=",  a || (b && c) ; puts " 4=",  a or (b and c)
puts " 5=", (a || b) && c  ; puts " 6=", (a or b) and c
#=============================================
```

```
D:\6w>log4
 1=
true
 2=
false
 3=
true
 4=
false
 5=
true
 6=
true
```

Рис. 3.3. Неправильна робота операторів and, or і not в операторах puts

```
#=================================================
if  false || true  && true  then  print " 1=",  "\n" end
if false or true and true   then  print " 2=",  "\n" end
if false || (true && true)  then  print " 3=",  "\n" end
if false or (true and true)  then  print " 4=",  "\n" end
if  (false || true) && true  then  print " 5=",  "\n"  end
if (false or true) and true  then  print " 6=",  "\n" end
#=================================================
```

```
D:\6w>log4
 1=
true
 2=
false
 3=
true
 4=
false
 5=
true
 6=
true
```

Рис. 3.4. Використання різних логічних операторів в умовному операторі

3.2. Порівняння логічних значень

Однією зі складових виконання дій з логічними значеннями true і false є операції порівняння (==, !=, >, <, > =, <=). В усіх мовах програмування встановлено співвідношення між цими логічними значеннями.

Наприклад, в мові `Object Pascal` приймається, що значення `false` = `0`, а значення `true` = `1`. Тому функція повернення номера елемента (`ord`) в послідовності даних порядкового типу (до яких відносяться крім цілих чисел також і логічні, символьні, перелічувані і тип-діапазон) `ord` (`false`) повертає значення `0`, а `ord` (`true`) повертає значення `1`.

Крім того, порівняння самих логічних значень `Object Pascal` дає:

```
true = true  #=> true (1)
true > true  #=> false (0)
true >= true  #=> true (1)
```

Для порівняння безпосередньо двох логічних значень в мові `Ruby` «працюють» тільки дві операції: дорівнює (= =) і не дорівнює (! =):

`true == true #=> дає наступний результат`

```
irb(main):006:0> true == true
=> true
```

`true >= true #=> дає наступний результат`

```
irb(main):002:0> true >= true
NoMethodError: undefined method `>=' for true:TrueClass
        from (irb):2
```

Такі результати можна бачити тому, що в `Ruby` справді не визначені методи (>, <,> =, <=) для роботи із логічними значеннями `true` і `false`. Але, при цьому, кожен об'єкт у створеній Юкіхіро Мацумото (автором мови `Ruby`) ієрархії має свій порядковий номер (т.зв. цілочисельний ідентифікатор – `integer identifier for obejct`), який може бути отриманий за допомогою методу `object_id`.

Нижче в табл. 3.4. наведені основні методи для отримання логічних об'єктів (значень) і роботи з ними. Слід пам'ятати, що методи, імена яких закінчуються знаком (?), повертають логічні значення `true` або `false`.

Таблиця 3.4.

Перелік основних методів, що застосовуються до логічних значень або повертають їх

Назва методу (символ операції, літерне позначення)	**Опис (назва операції, оператора)**
`! (not)`	Заперечення значення.
`&& (and)`	Логічне множення
`|| (or)`	Операція АБО.
`^ (xor)`	Операція виключаюче Або.

Назва методу (символ операції, літерне позначення)	Опис (назва операції, оператора)

`==`	Метод Дорівнює. Повертає true, якщо значення першого числа дорівнює значенню другого числа.
	`1 == 1 #=> true  -чи дорівнює?`
	`3 == 1 #=> false -чи дорівнює?`

`!=`	Метод Не дорівнює. Повертає true, якщо значення першого числа не дорівнює значенню другого числа. 3! = 2 # => true - Більше? 3! = 3 # => false - Більше?
`>`	Метод Більше. Повертає true, якщо значення першого числа більше значення другого числа. 3> 2 # => true -Більше? 1> 3 # => false -Більше?
`<`	Метод Менше. Повертає true, якщо значення першого числа менше значення другого числа. 1 <2 # => true -Менше? 7 <2 # => false -Менше?
`>=`	Метод Більше або дорівнює. Повертає true, якщо значення першого числа більше або дорівнює значенню другого числа. 9> = 2 # => true -Менше або дорівнює? 1> = 1 # => true -Менше або дорівнює? 4> = 7 # => false -Менше або дорівнює?
`<=`	Метод Менше або дорінює. Повертає true, якщо значення першого числа менше або дорівнює значенню другого числа. 1 <= 2 # => true -Менше або дорівнює? 1 <= 1 # => true -Менше або дорівнює? 4 <= 1 # => false -Менше або дорівнює?
`.eql?`	Метод повертає true, якщо обидва порівнюваних об'єкта одного і того ж типу і мають однакові значення. 1 == 1.0 # -> true 1.eql? (1.0) # -> false (1.0) .eql? (1.0) # -> true
`.integer?`	Повертає `true`, якщо об’єкт є цілим числом.
`.odd?` `.even?`	Методи повертають true, якщо число відповідно непарне і парне і false, якщо навпаки. 1.even? # => False 1.odd? # => True 2.even? # => True 2.odd? # => False

Назва методу (символ операції, літерне позначення)	Опис (назва операції, оператора)
`.zero?`	Метод перевіряє, чи є число нулем. Якщо так, то повертає true, інакше false. 0.zero? # => True 4.zero? # => False
`.nonzero?`	Метод повертає саме число, якщо воно не є нулем, якщо число - нуль, то повертає значення nil, що оцінюється як false. 0.nonzero? # => Nil 4.nonzero? # => 4
`<=>` (fix `<=>` numeric)	Метод <=> використовується для порівняння двох чисел a і b і повертає 0, 1, -1 - відповідно, якщо a == b, якщо a> b і якщо a <b. 25 <=> 27 # => -1 25 <=> 25 # => 0 25 <=> 3 # => 1

3.3. Формування рядків при друкуванні

Рядки тексту в `Ruby` обмежуються або одинарними (`'.'`), Або подвійними лапками (`".`"). Однак такі рядки мають між собою суттєві відмінності. При використанні подвійних лапок допускається вставляти в рядки керуючі `escape`-послідовності, що починаються з символу `'\'`, а також обчислювати вирази за допомогою конструкції `# {}`. наприклад:

```
print "\n"
print '\n'
print "\001","\n"
print '\001',"\n"
print "abcd #{5*3} efg","\n"
var = " abc "
print "1234#{var*3}567","\n"
```

В результаті виконання даного коду будуть виведені на екран наступні символи:

```
< порожній рядок, новий рядок >
\n
\001
abcd 15 efg
1234 abc abc abc 567
```

3.4. Оператори розгалуження

Оператор розгалуження (умовний оператор, оператор вибору), дозволяє в залежності від істинності чи хибності деякого умови виконувати ту чи іншу послідовність команд.

Саме умова записується у вигляді логічного висловлювання (логічного виразу) виду: `A <операція порівняння> B.`

Наприклад: `A == B`.

В `Ruby` операції порівняння називаються методами. `Ruby` підтримує всі конструкції вибору, які традиційно використовуються в інших мовах програмування. Але перш ніж говорити про них, зупинимося на деяких особливостях `Ruby` при обчисленні істинності окремих виразів.

Найпростіша конструкція умовного оператора має вигляд `if..then..end`. Вона записується у вигляді:

```
if “умова” then
     “оператор виконання умови”
else
     “код, який виконується в разі невиконання умови”
end
```

Оператор `if` повинен закриватися операторної дужки `end`, як це прийнято робити в мові `Visual Basic for Application`. В `Ruby` ключове слово `then` можна опускати. Слід запам'ятати, що тільки `false` і `nil` трактуються як помилкові значення в логічному контексті. Всі числові значення, є істинними. Рядки, взагалі кажучи, не є для умовних операторів істиною. Для того щоб краще зрозуміти сенс сказаного вище, рекомендується виконати наступний код:

```
if 23
  puts "23 is true" # такий же результат дасть значення 2.3
end

if “qq”
  puts "All strings is true"
end # видасть попередження: string literal in condition
```

`Ruby` підтримує всі стандартні логічні оператори, а також додатковий оператор `defined?`. Даний оператор повертає `nil`, якщо його аргумент не визначений, і його опис в іншому випадку. Нижче наведені приклади застосування цього оператора.

```
puts defined? 1      #=> повертає expression
puts defined? dummy  #=> повертає «пусто»
puts defined? printf #=> повертає method
puts defined? c = 1  #=> повертає assignment
puts defined? 42.abs #=> повертає method
```

Оператор `if` також може використовуватися в правій частині оператора присвоювання, наприклад:

```
str = if i < 100 then "Мало" else "Побольше" end
puts str
```

Крім того, в умовних операторах може застосовуватися ключове слово `unless`, наприклад:

```
unless x >=5 then
     statement1
else
     statement2
end

statmement1 unless y != 3
x = unless a<= 0 then else b end
```

При виборі з більшого числа альтернатив (множині виборі) зручніше використовувати оператор `case..when`. Основною його перевагою є те, що він працює значно швидше оператора `if..then..end`.

```
case <селектор>
when <варіант1> then значення
    ...
when <варіант2> then значення
    ...
when  <тестN>   then значення
    ...
else                  значення
end
```

В операторі `case <селектор>` послідовно порівнюється з виразами `<варіант>` до тих пір, поки, воно не співпаде з одним з них (порівняння проводиться за допомогою операції `==`), після чого виконується відповідний фрагмент коду. Оператор `case` повертає значення останнього обчисленого виразу, або nil, якщо такого не було. наприклад:

```
scale = 8
case  scale
when    0 then  puts "lowest"
when 1..3 then  puts "medium-low"
when 4..5 then  puts "medium"
when 6..7 then  puts "medium-high"
when 8..9 then  puts "high"
when   10 then  puts "highest"
else       puts "off scale"
end
# => high
```

Тут діапазон `1..3` означає діапазон чисел від `1` до `3`, включно. Так як значення `scale` дорівнює `8`, `scale` відповідає діапазону `8..9` і `scale` повертає рядок `"high"`. Але якщо використовувати в діапазоні три точки (наприклад,

`1 ... 5`), кінцеве значення `5` виключається. Сукупності точок `'..'` і `'...'` називаються операторами діапазону. Негласно оператор `case` використовує оператор `'==='` з `Range`, який перевіряє, чи є значення членом діапазону.

3.5. Оператори циклу і методи-ітератори

Для завдання повторюваних дій в більшості мов програмування використовуються оператори циклу. В Ruby для цих цілей є кілька операторів циклу і безліч методів-ітераторів, що включають в свою структуру так звані блоки. Цикли і ітератори, як правило, використовуються для обробки елементів колекцій. Для кожного класу об'єктів даних (Array, String і ін.), з яких складаються колекції, існує свій набір методів-ітераторів, які широко використовують різні конструкції блоків. Колекцією називається будь-який набір елементів. Вона представляє собою об'єкт певного типу (екземпляр певного класу), який складається з інших об'єктів. Наприклад, об'єкт масив з ім'ям `a = [ "1", "2", "3"]` (що належить класу Array) містить колекцію символів (що належать класу String).

Якщо колекція містить в собі інші колекції, то така колекція називається розгалуженою колекцією, деревом або Root-колекцією (кореневою колекцією), так як від неї починає відбуватися розгалуження. Двома найбільш важливими колекціями в Ruby є масиви і хеши.

3.6. Блоки циклів і ітераторів

Цикли і ітератори призначені для неодноразового виконання різноманітних фрагментів коду. В Ruby такі фрагменти коду оформляються у вигляді блоків.

Розглянемо фрагмент програми:

```
animals = %w( ant bee cat dog elk )
animals.each {|animal| puts animal }# виведення: ant bee cat dog elk
```

В останньому рядку each – це метод класу Array, призначений для послідовної передачі елементів масиву в блок. Власне блоком є вираз:

```
{|animal| puts animal }
```

Таким чином, блоком називається довільний код, який можна передати будь-якому методу як неявний аргумент останнього. Слід розуміти, що при цьому блок є особливою конструкцією мови та відокремлений від списку явних аргументів методу, що означає наступне:

• методу можна передати лише один блок;

• блок завжди розташовується в самому кінці і винесено за дужки з аргументами (якщо вони є);

• блок можна задати при виклику будь-якого методу, незалежно від того, вказані у визначенні методу аргументи чи ні.

Код блоку може перебувати всередині фігурних дужок {} або ключових слів do end. Нижче наведені приклади простих блоків, які виводять текст "Hello world!":

```
{ puts "Hello world!" }
do puts "Hello world!" end
```

Цикли є операторами, які неодноразово виконують наступні за ними блоки.

Ітератори – це методи, які приймають і виконують блоки або об'єкти класу `Proc`. Як правило, блок розташовується безпосередньо після виклику методу. Ітератори використовуються для визначення користувачем керуючих структур, а найчастіше – циклів.

Загальний вигляд конструкції циклів і ітераторів:

```
<оператор циклу> <тіло блоку>
<об'єкт. імя_метода_ітератора> <тіло блоку>
<об'єкт_1> = <об'єкт_2.імя_метода_ітератора> <тіло блоку>
```

При цьому:

– тіло блоку оператора циклу обов'язково розташоване в операторних дужках (контейнері) – `do end`.

– тіло блоку ітератора обов'язково розташоване в операторних дужках (контейнері) – `{}`.

Нижче наведені приклади правильного і неправильного використання операторних дужок блоків `do end` і `{}` в операторі циклу `for..do` і методі-ітераторі `each_line`.

```
for i in 1..10 do print "#{i} " end #=> виведення 1 2 3 4 5 6 7 8 9 10 в рядок
for i in 1..10 { print "#{i} " } #=> виведе помилку
"a\nb\nc\n".each_line { |k| print k } #=> виведе a b c в стовпчик
"a\nb\nc\n".each_line do |k| print k end #=> виведе помилку
```

3.7. Оператор циклу з параметром for..do

Синтаксис циклу `for..do` наступний (квадратні дужки означають, що елемент циклу `do` не є обов'язковим):

for *параметр* in *нижня_границя_діапазону* .. *верхня_границя_діапазону* [do]
    тіло циклу
end

Приклад циклу з використанням елемента `do`:

```
for k in 1..10000 do
        вираз
end
```

Приклад використання конструкції `for..do` для виведення чисельних значень без елемента `do`:

```
# цикл за значеннями параметра циклу:
for i in 1..10
print "#{i} "
end

# цикл за значеннями в масиві:
for value in [1,2,3,4,5,6,7,8,9,10]
print "#{value} "
end
```

Обидва приклади програмних фрагментів виводять числа від 1 до 10.

3.8. Оператор циклу з передумовою while

Цикл `while` - це цикл з передумовою, тобто оператори, які складають його тіло виконуються нуль або більше разів. Дія циклу триває до тих пір, поки істинна його умова (тобто поки значення логічного виразу, що описує умова дорівнює true (істина)). Друга частина оператора циклу while - зарезервоване слово do може бути опущено.

while *умова* [do]
    вираз
end

Приклад використання циклу `while` з та без слова `do`:

```
k = 1
while k <= 10
k = k + 1 # можно записать k+=1
if k < 10 then puts k end
end
```

```
k = 1
   while k <= 10 do
   k = k+=1
   if k < 10 then puts k end
   end
```

3.9. Оператор циклу з умовою поста (while або until)

Даний вид циклу може містити оператори while або until. Для виконання циклу while умова має бути істинно, для циклу until - помилково.

Дана конструкція виконується поки умова ВІРНА (значення умови дорівнює true), але тіло самого циклу виконується при будь-якій умові хоча б один раз. Синтаксис циклу з оператором `while`:

```
begin
   тіло циклу
end  while  умова
```

Приклад коду:

```
i=1
while i <= 10
  puts i; i += 1
end #=> виведе 1 2 3 4 5 6 7 8 9 10 у стовпчик
```

Логіка його виконання така сама, як і у циклу `while`, з однією важливою відмінністю – цикл виконується до тих пір, поки умова НЕ ВІРНА (значення умови дорівнює `false`). Тут також можна опускати зарезервоване слово `do`. Синтаксис циклу з оператором `until`:

```
until  умова  [do}
тіло циклу
end
```

Приклад коду:

```
i=1
until i > 10
print "#{i} "
i+=1
end #=> виведе 1 2 3 4 5 6 7 8 9 10 у рядок
```

3.10. Використання циклів while та until в якості модифікаторів

Ruby також дозволяє використовувати ключові слова while і until в якості модифікаторів. Це означає, що програміст може використовувати їх *постфіксну* форму, тобто ставити заголовок циклу після висловлення, на відміну від звичайної *префиксной* форми. Фактично блоки коду подаються на їх вхід. Завдяки цьому Ruby надає можливість писати дуже короткі цикли без втрати читабельності коду, наприклад:

```
#Перший приклад:
i=0
print "#{i+=1} " while i < 10
#Другий приклад:
i=0
print "#{i+=1} " until i == 10
```

Обидва приклади виведуть числа від 1 до 10 у рядок.

3.11. Цикл loop

Цикл loop це один з простих циклів, який може містити зарезервовані (службові) слова: break, redo, next і retry, які змінюють звичайний порядок виконання циклів. Їх дія описано в наступній таблиці 5.

Таблиця 3.5.

Оператори виходу з циклів

Оператор	Вуконувані дії
break	Негайно припиняє виконання циклу; управління передається на затвердження, розташоване відразу за циклом
redo	Повторює тіло циклу з початку, не перераховуючи умова виконання циклу (не переходячи до наступного елементу в разі ітератора)
next	Пропускає частину тіла циклу, яка йде за ним, і переходить до наступної ітерації
retry	Починає виконання циклу з самого початку

Ключове слово break дозволяє вийти з циклу в будь-який необхідний момент, наприклад:

```
loop do
i+=1
print "#{i}"
break if i==10
end
```

Даний код виведе числову послідовність: 1 2 3 4 5 6 7 8 9 10.

Таким чином, в даному прикладі цикл припиняє роботу після того, як змінна і прийме значення, що дорівнює 10.

Ключове слово `next` використовується для пропуску поточної ітерації і переходу до наступної, наприклад:

```
i=0
loop do
i+=1
next if i==5
print "#{i} "
break if i==10
end
```

Цей код виведе числа від 1 до 10, але пропустить число 5: 1 2 3 4 6 7 8 9 10.

Вищевказані інструкції `break`, `redo`, `next` і `retry` можна використовувати і в інших циклах. Нижче в таблиці 3.6 наведено приклад використання зазначених інструкцій у ітераторі `for`.

Таблиця 3.6.

Приклади використання різних операторів виходу із циклу

Код програми	Результати

`for i in 1 .. 5`
`  print  i`
`  break if i == 3`
`  print "*"`
`end`
	`  Результат:`
	`1*2*3`

`for i in 1 .. 5`
`  print  i`
`  redo if i == 3`
`  print "*"`
`end`
	`  Результат:`
	`1*2*33333 ...`
	`Виконання циклу не завершується`

`for i in 1 .. 5`
`  print  i`
`  next if i == 3`
`  print "*"`
`end`
	`  Результат:`
	`1*2*34*5*`

`for i in 1 .. 5`
`  print  i`
`  retry if i == 3`
`  print "*"`
`end`
	`  Результат:`
	`1*2*31*2*31*2*...`
	`Виконання циклу не завершується`

3.12. Методи-ітератори each, each line, times, upto і step

Ітератор each

Як зазначалося вище, *ітератори* (від англ. `iterate` – повторювати) – це методи, які приймають на вхід блоки і неодноразово виконують код, що міститься у цих блоках. Найбільш часто вони застосовуються для обробки елементів колекцій (масивів або хешів). Загальна конструкція ітератора така:

```
масив_1.метод_і {|індекс| блок} => результат - масив_2
```

Метод `метод_і` виконує вираз *`блок`* для кожного елемента масиву `масив_1`. Таким чином створюється новий `масив_2`, що складається зі значень, які отримані при обчисленні виразу *`блок`*. На кожному кроці поточний елемент масиву `массів_1` присвоюється змінній *`індекс`*. Для будь-яких інших типів колекцій ітератор можна уявити так:

<об'єкт>. <назва_метода_ітератора> {|змінна| <виконуваний_оператор>}

Об'єктом є будь-яка сутність, яка обробляється. Для об'єктів різних класів існують різні ітератори. Наприклад, для об'єктів класу `String` застосовуються ітератори `.each_byte` і `each_line`. У фігурні дужки блоку беруть два об'єкти:

– *змінна* (на місце якої підставляються елементи оброблюваної послідовності сутностей колекції);

– *оператор*, фрагмент коду, який застосовується до ініціалізованої *змінної*.

Наприклад, нижченаведений ітератор `.each_line` обробляє об'єкт, який представляє собою рядок символів. На місце змінної `k` підставляється послідовно кожен символ з оброблюваного рядка, а потім до неї застосовується оператор `print k`. В результаті на екран виводиться стовпець символів.

Символи "a", "\ n", "b", "\ n", "c", "\ n" розглядаються як окремі елементи, з огляду на те, що сукупність символів "\ n" є керуючим прапором переходу на новий рядок (`line`).

```
"a\nb\nc\n".each_line{|k| print k}
#=> результат виконання
a
b
c
```

Таблиця 3.7.

Послідовність кроків виконання інтерпретатора each_line

№ кроку	Виконувана дія
1 крок	метод `each_line` вибирає перший елемент колекції елементів класу `String` – "a\n". При цьому враховується, що рядок укладений в подвійні лапки, в яких можна використовувати керуючі символи – в даному випадку перевід рядка \n.
2 крок	змінній k призначається значення "a\n"
3 крок	Вираз (оператор) `print k` виводить значення елемента колекції (рядки) "a" на екран і переводить покажчик позиції курсора на початок нового рядка
4 крок	метод `each_line` обирає другий елемент колекції елементів класу `String` - "b\n". При цьому також враховується, що рядок поміщений в подвійні лапки, в яких можна використовувати керуючі символи - в даному випадку перехід рядка \n.
... Послідовність дій триває до вичерпання списку елементів вхідної колекції. В результаті буду виведені три символи: a b c	

Графічно роботу ітератора можна представити у такий спосіб:

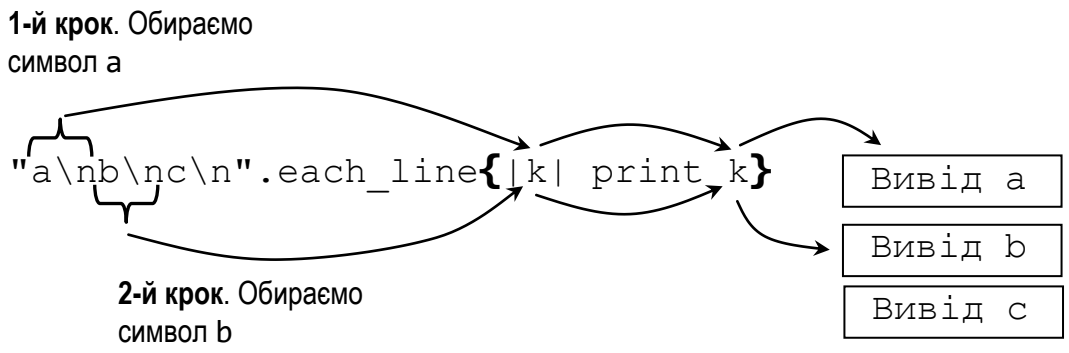


Рис. 3.1. Графічне представлення роботи ітератора `each_line`

Наступний найпростіший метод-ітератор - це `each`. Він виконує код блоку для кожного елемента вхідного масиву, передаючи в блок поточний елемент як параметр. Даний итератор змінює значення вихідного масиву, тому бажано застосовувати метод з аналогічною дією reject.

Наприклад, можна перебрати елементи масиву і при цьому кожен його елемент передавати в блок, який буде їх просто друкувати (виводити) на екран:

```
[1,2,3,4,5,6,7,8,9,10].each {|value| print "#{value} "}
  #=> будуть виведені значення 1 2 3 4 5 6 7 8 9 10
  a = [ "a", "b", "c" ]
  a.each {|x| print x, " -- " }  #=>   результат:
  a -- b -- c –
```

Ітератор times

`Times` – це метод класу `Integer`, тобто цілих чисел. Ітератор `times` аналогічний циклу `for` в інших мовах і дозволяє виконувати цикл і здійснювати певні операції, які передаються йому в блоці N разів. Наприклад, наведений нижче код десять разів виконує розташований за ним оператор `print`, починаючи з `i = 0`.

```
10.times {|i| print "#{i} "}
#=> результат виконання 1 2 3 4 5 6 7 8 9 10
```

Ітератори upto і step

Ітератор `upto` аналогічний циклу `for` з тією лише відмінністю від ітератора `times`, що в ньому є можливість задавати стартову позицію (в `times` старт починається з нуля).

```
1.upto(10) {|i| print "#{i} "}
#=> результат виконання 1 2 3 4 5 6 7 8 9 10
0.upto(9) do |x| print x, " " end
#=> результат виконання 0 1 2 3 4 5 6 7 8 9
```

Ітератори також дозволяють задавати крок для кожної ітерації за допомогою оператора (ітератора) `step`:

```
step(10, 2) { |i| print "#{i} "}
```

Итератор `step` еквівалентний циклу `for` з інкрементом рівним `2`, тобто при кожній ітерації параметр `i` збільшується на `2`. Результат виконання вищенаведеного ітератора `step` буде виведений на екран в наступному вигляді `1 3 5 7 9`.

Ітератор each_index

Іноді в заданому масиві необхідно проводити ітерації не по значенням, а за індексами масиву. Для цих цілей і призначені ітератори `each` та `each_index`. Записати звернення до ітератора `each` можна так:

```
[1, 1, 2, 3].each {|k| print k, " " } # результат: 1 1 2 3
```

Звернення до оператора `each_index`

```
array = [10,20,30,40,50,60,70,80,90,100]
array.each_index {|i| print "#{array[i]} "}
 #=> результат:10 20 30 40 50 60 70 80 90 100
```

Блок приймає індекси елементів масиву і його можна використовувати для виведення значень елементів.

3.13. Завдання до лабораторної роботи

Завдання для програмування береться з файлу додатку до Лабораторних робіт. Номер варіанта відповідає номеру комп'ютера, на якому працює студент. Звіт до даної лабораторної роботи повинен містити.

1. Назву роботи.
2. Тему, мету і завдання до роботи.
3. Вихідний код збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

ЛАБОРАТОРНА РОБОТА №4.

Масиви, вектори і матриці

Тема: Використання одновимірних і багатовимірних масивів у вирішенні математичних задач.

Мета: Використання мовних конструкцій мови Ruby для написання і налагодження програм.

Теоретичні відомості:

4.1. Опис і заповнення одновимірних масивів

Слід відзначити, що для зберігання і обробки індексованих (викликаються за номером індексу або ключу) даних (об'єктів) в `Ruby` застосовуються такі структури:

– Масиви – екземпляри класу `Array`.

– Матриці – екземпляри класу `Matrix`.

– Вектори – екземпляри класу `Vector` (підключається одночасно з матрицями (клас `Matrix`)).

– Рядки – екземпляри класу `String`.

– Хеши – екземпляри класу `Hash`. У даній лабораторній роботі рядки і хеші не розглядаються.

Кожен з перерахованих вище класів має свої методи для обробки збережених об'єктів.

Масив – це впорядкована колекція довільних об'єктів з целочисленной індексацією. Наприклад, запис виду: `b[0] = 7`, визначає масив `b`, перший елемент якого дорівнює `7`. Таким чином, дані, представлені в масиві, мають парну структуру «індекс – значення», в якій індекс служить для ідентифікації елемента в масиві і може бути тільки цілим числом. Слід зауважити, що ключів (індексів) у кожного елемента масиву є *цілих два*: *лівобічний* і *правобічний*, як і у строк. Нумерація елементів масиву починається з `0`, як у мовах `C` або `Java`. Негативний індекс передбачає відлік з кінця масиву, тобто індексу `-1` відповідає останній елемент масиву, `-2` – передостанній, і так далі. Наприклад, для згаданого вище одноелементного масиву `b` елемент `b[-1]` теж дорівнює `7`.

Для трьохелементного масива `c[10,20,30]` значення індексів елементів такі:

```
 -3  -2  -1
  ↓   ↓   ↓
c[10, 20, 30]
  ↑   ↑   ↑
  0   1   2
```

тобто

```
c[0]=c[-3]  дорівнює 10
c[1]=c[-2]  дорівнює 20
c[2]=c[-1]  дорівнює 30
```

Методи класу `Array` – `[]` і `new` - призначені для створення масивів. До методів обробки об'єктів класу відносять наступні: `[]=`, `[]`, `&`, `|`, `*`, `+`, `-`, `<<`, `<=>`, `==`, `abbrev`, `assoc`, `at`, `clear`, `collect!`, `collect`, `compact!`, `compact`, `concat`, `delete_at`, `delete_if`, `delete`, `each_index`, `each`, `empty?`, `eql?`, `fetch`, `fill`, `first`, `flatten!`, `flatten`, `frozen?`, `hash`, `include?`, `indexes`, `index`, `indices`, `insert`, `inspect`, `join`, `last`, `length`, `map!`, `map`, `nitems`, `pack`, `pop`, `push`,

rassoc, reject!, reject, replace, reverse!, reverse_each, reverse, rindex, select, shift, size, slice!, slice, sort!, sort, to_ary, to_a, to_s, transpose, uniq!, uniq, unshift, values_at, zip.

Коли потрібно створити масив, але не заповнювати його елементами використовують методи [ ] або new.

```
a = [ ]        #=> пустий масив a
b = Array.new #=> пустий масив b
```

Також можна описувати об'єкт типа Range (діапазон) та визивати метод .to_a.

```
b = (1..6).to_a    #=> [1, 2, 3, 4, 5, 6]
```

Ще один метод створення (непорожнього) примірника класу Array, полягає у вказівці його елементів, розділених комами, в квадратних дужках, наприклад:

```
a = [1, 2, 3]    #=> масив a який складається з 3-х цілочисельних об'єктів
```

Використання методу .new для створення непорожньої масиву приведено нижче. Необов'язковий другий аргумент даного методу задає клас, екземпляри якого заповнюють створюваний масив.

```
Time = 6
timeArray = Array.new(3, Time)
puts timeArray            #=> [6, 6, 6]
t_arr = Array.new(3, 2)
puts t_arr                #=> [2, 2, 2]
x = Array.new(6){ |index| index + 1 }
puts x                    #=> [1, 2, 3, 4, 5, 6]
```

У передостанньому операторі параметром методу .new є кількість елементів майбутнього масиву (це число 6). У фігурних дужках міститься блок, що заповнює масив. Вбудований оператор index + 1 визначає, що кожне значення елемента масиву буде на одиницю більше його індексу.

Масиви в Ruby можуть включати в себе елементи різних типів (класів). наприклад:

```
ary = [1, 2, "3"]
ary1 = ary + ["foo", "bar"]   #=> [1, 2, "3", "foo", "bar"]
ary2 = ary * 2                #=> [1, 2, "3", 1, 2, "3"]
```

4.2. Приклади використання масивів

Нехай потрібно ввести з клавіатури масив чисел і надрукувати суму всіх елементів. Наведемо два рішення цього завдання.

У першому рішенні спочатку вводиться кількість елементів масиву, а потім його елементи заповнюються значеннями, одним за іншим. Зверніть увагу, що перший елемент масиву має індекс 0, а останній – на одиницю менший, ніж розмірність масиву.

```
print "Vvedite chislo elementov massiva: "
sn = gets.chop!
   n = sn.to_i
b = Array.new(n)   # створення масиву з n елементів
s = 0              # обнуління суму
for i in 0 .. n - 1
  print "Vvedite #{i+1}-y element massiva: "
  b[i] = gets.chop!.to_f
       s = s + b[i]
end
print "Summa vseh elementov massiva ravna ", s, "\n"
```

У другій версії заданої програми, що приведена нижче, всі числа вводяться відразу у вигляді одного рядка, в якій дані відділяються один від одного пробілами (наприклад, `23 -34.67 100.5`).

Вбудований метод `split` розділяє рядок на елементи масиву, аргументом цього методу є роздільник (якщо роздільником є пробіл, то можна викликати цей метод без аргументу). У разі, якщо б знадобилося розділяти числа, наприклад, крапкою з комою, то виклик методу виглядав би так:

`a.split (';')`.

Для визначення довжини масиву застосувується метод `length` (який можна замінити на еквівалентну їй метод `size`).

```
puts "Vvedite massiv chisel razdelja ih probelami:"
a = gets.chop!
b = a.split    # разбили строку на отдельные числа
s = 0
for i in 0 .. b.length - 1
  s += b[i].to_f
end
puts "Summa vseh elementov missiva ravna #{s}"
```

Результат роботи програми

```
D:\6w>array1
Vvedite massiv chisel razdelja ih probelami:
2 34 45 -23 38 21
Summa vseh elementov missiva ravna 117.0
```

4.3. Застосування циклів і ітераторів для обробки масивів

Розглянемо застосування різних типів *циклів* і *ітераторів* для обробки масивів.

Як відомо, ітератори є не чим іншим, як методами, що підтримують колекції. Колекціями називаються об'єкти, які зберігають в собі групи елементів однотипних або різнотипних даних.

Ітератори – це методи, об'єднані з блоками, які послідовно переглядають всі елементи колекції (масиву або хеша) змінюючи кожен з них відповідно до визначеного в блоці алгоритмом дій.

Для демонстрації можливостей циклів і ітераторів визначимо масив `list`, що складається з п'яти текстових елементів:

```
list = %w[alpha bravo charlie delta echo];
p list  #=>  выводит элементы массива в кавычках
```

Таким чином, вміст масиву при виведенні має наступний вигляд:

```
D:\6w>matr3
["alpha", "bravo", "charlie", "delta", "echo"]
```

У наведених у таблиці 4.1. прикладах *цикли* і *ітератори* використовуються для виведення на екран всіх елементів (значень) масиву `list`.

Таблиця 4.1.

Цикли і ітератори, що виводять на екран вміст масиву `list`

# Цикл 1 (while)	**# Цикл 2 (until)**
`i=0 ; while i < list.size do print "#{list[i]} " ; i += 1 end`	`i=0 ; until i == list.size do print "#{list[i]} " ; i += 1 end`
# Цикл 3 (for)	**# Цикл 4 (итератор 'each')**
`for x in list do print "#{x} " end`	`list.each do |x| print "#{x} " end`
# Цикл 5 (метод 'loop')	**# Цикл 6 (метод 'loop')**
`i = 0 ; n=list.size-1 ; loop do print "#{list[i]} " ; i += 1 break if i > n end`	`i=0 ; n=list.size-1 ; loop do print "#{list[i]} " i += 1 ; break unless i <= n end`
# Цикл 7 (итератор 'times')	**# Цикл 8 (итератор 'upto')**
`n=list.size ; n.times do |i| print "#{list[i]} " end`	`n=list.size-1 0.upto(n) do |i| print "#{list[i]} " end`
# Цикл 9 (for)	**# Цикл 10 ('each_index')**
`n=list.size-1 for i in 0..n do print "#{list[i]} " end`	`list.each_index do |x| print "#{list[x]} " end`

4.4. Використання методів для роботи з масивами

Одним з безперечних переваг Ruby є наявність в ньому великої кількості *методів* для роботи з масивами.

Слід особливо відзначити, що для підвищення функціональності до методів приєднуються так звані *блоки*. Ці блоки часто називають ітераторами, так як вони формують алгоритми послідовностей необхідних дій, які виконуються з елементами масивів. Методи з ітераторами називають методами-ітераторами.

Блок являє собою фрагмент коду специфічного виду укладений в операторні дужки (контейнер) `do ... end` або фігурні дужки `{}`. Як правило, блок повертає значення останньої виконаної інструкції. Такий результат може бути масивом, дійсним значенням і т.д. В цілому структура блоку для методу-ітератора має вигляд:

```
{ | elm | expression } или
do | elm | expression
  end
```

Крім того, в Ruby часто використовуються методи зі знаком питання в кінці імені. Такі методи зі знаком питання традиційно працюють як предикати, тобто повертають значення `true` або `false`.

Нижче в таблиці 4.2. перераховані найбільш часто використовувані методи, що дають можливість роботи з масивом як з безліччю, стеком і іншими структурами даних. Більш детально інформація про методи приведена в документі «Ruby Довідник Array».

Таблиця 4.2.

Найбільш часто використовувані методи для об'єктів класу Array

Призначення і приклад використання методу		Результат

[ ]
at
	Отримання елемента зі вказаним індексом. Якщо аргумент є негативним числом, то індекс починає свій відлік з кінця послідовності елементів массиву
	`a = ["a", "b", "c", "d"];`
	`puts a[0]; puts a.at(0)`
	`puts a[0 .. 2]; puts a[-2];`
	`puts a.at(-2); puts a[2, 2]`
		`"a"; "a"`
		`["a", "b", "c"]; "c"`
		`"c"; ["c", "d"]`

+	Додавання одного масива до іншого `[1, 2, 3] + [4, 5]`	`[1, 2, 3, 4, 5]`
*****	Повторення `[1, 2] * 2`	`[1, 2, 1, 2]`

Продовження таблиці 4.2.

|	Об’єднання множин `["a", "b", "c"] | ["c", "d", "a"]`	`["a", "b", "c", "d"]`
&	Перетин множин `[1, 1, 3, 5] & [1, 2, 3]`	`[1, 3]`
-	Різність множин `[1, 2, 2, 3, 3, 3, 4, 5] - [1, 2, 4]`	`[3, 5]`

collect
map
	Виконує вираз в блоці для кожного елемента масиву. Створює новий масив, що складається з значень, які отримані при обчисленні виразу в блоці.
	`b = [1, 2, 3].map do |i|  i + 1 end #=> b =`
	`c = [1, 2, 3].collect { |i|  i + 1 }`
	`puts b;  puts c`
	`d = 2; d_mul_c = b.map{ | elem | elem * b }`
		`[2, 3, 4]`
		`[2, 3, 4]`
		`[4, 6, 8]`

pop	Видалення останнього елемента масиву
	`a = [ "a", "m", "z" ]; a.pop`
	`puts a`
		`a m`

<<
push
	Додавання в кінець масиву
	`[1, 2] << "c" << [3, 4]`
	`a = ["a", "b"]; a.push("c", "d")`
		`[1, 2, "c", [3, 4]]`
		["a", "b", "c", "d"]

unshift	Додавання елемента в початок масиву із зсувом інших `a = ["b", "c"]; a.unshift("a")`	["a", "b", "c"]
clear	Видалення всіх елементів масиву `a = ["a", "b", "c", "d"]; a.clear`	`[]`

collect	Виконання блоку операторів, що розміщені у фігурних дужках, по разу для кожного елемента масиву
	`a = ["a", "b", "c", "d"]`
	`a.collect {|i| i+"!"}`
		["a!", "b!", "c!", "d!"]

compact	Видалення всіх об'єктів `nil` з масиву `["a", "b", nil, "c", nil].compact`	`["a", "b", "c"]`

each	Виконує код в блоці для кожного елемента масиву, передаючи в блок поточний елемент як параметр. Здійснює виклик блоку операторів по одному разу для кожного елемента масиву
	`a = ["a", "b", "c"]`
	`a.each {|x| print x, "--" }`
	`ary = [1,2,3,4,5]; ary.each do |i| puts i end`
		a--b--c—
		`1 2 3 4 5`

each_index	Працює так, як і `each`, але передає в блок не поточний елемент, а індекс поточного елемента.
	`a = [ "a", "b", "c" ]`
	`a.each_index {|x| print x, " -- " }`
	#=> результат: `0 -- 1 -- 2 --`
		`0 -- 1 -- 2 --`

length	Визначення кількості елементів масиву `puts [ 1, 2, 3, 4, 5 ].length`	`5`

max	Пошук максимального елемента
	`c = [2, -7, 8, 23]; puts c.max`
	`b = [ "a", "min", "zero" ]; puts c.max`
	`puts c.max ; puts b.max`
		`23`
		`zero`
		`23 “zero”`

min
min_by
	Пошук мінімального елемента
	`c = [2, -7, 8, 23] ; puts c.min`
	`b = [ "a", "min", "zero" ]; puts b.min`
	`puts c.min_by{ |elem| elem.size }`
	`puts b.min_by{ |elem| elem.size }`
		`-7`
		`a`
		`-7 a`

Продовження таблиці 4.2.

first	Повертає перший елемент або перші n елементів масиву. Якщо масив порожній, то для першої форми виклику (без n) повертається nil, а для другої – порожній масив.
	`a = [ "q", "r", "s", "t" ]`
	`a.first #=>`
	`a.first(1) #=>`
	`a.first(3) #=>`
		`"q"`
		`["q"]`
		`["q", "r", "s"]`

flatten	"Розгладження" масиву – отримання одновимірного масиву з масиву масивів
	`s = [ 1, 2, 3 ]`
	`t = [ 4, 5, 6, [7, 8] ]`
	`a = [ s, t, 9, 10 ]; a.flatten`
		[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

join	Перетворення масиву в рядок
	`puts [ "a", "b", "c" ].join`
	`puts [ "a", "b", "c" ].join("-")`
	`puts [1, 2, "3"].join(":")`
		`"abc"`
		`"a-b-c"`
		`"1:2:3"`

index	Отримання номера першого входження аргументу (nil якщо не знайдене)
	`a = [ "a", "b", "c" ]; puts a.index("b")`
	`puts a.index("z")`
		`1`
		`nil`

reverse	Інвертування масиву
	`puts ["a", "b", "c"].reverse`
	`puts [ 1 ].reverse`
		`["c", "b", "a"]`
		`1`

sort	Сортування (впорядкування) масиву за зростанням
	`a = ["d", "a", "c"]`
	`puts a.sort`
	`puts a.sort {|x,y| y <=> x }`
		`["a", "c", "d"]`
		`["d", "c", "a"]`

sort. reverse	Відсортувати і інвертувати масив. Змінює масив.
	`a = ["d", "a", "c"]`
	`puts a.sort.reverse`
		`d c a`

size	Визначення кількості елементів в масиві
	`puts [ "a", "b", "c" ].size`
	`puts [ 1 ].size`
	`puts [1, "считайте", 3, "количество", 5, 6, "запятых", 2, 5].size`
		`3`
		`1`
		`9`

include?(6)	З'ясувати, чи є в масиві елемент зі значенням 6 `a = [6,3,5,1,2,4]; puts a.include?(6)`	`true`

empty?	З'ясувує, порожній чи масив
	`a = [6,3,5,1,2,4]; puts a.empty?`
	`[].empty? #=>`
		`false`
		`true`

eny?	Відсортувати і інвертувати масив `a = [6,3,5,1,2,4]; puts a.sort.reverse`	

delete	Видалити елемент зі значенням 6. Змінює масив.
	`a = [6,3,5,1,2,4]; a.delete(6)`
	`puts a`
	`b = [ "a", "b", "b", "b", "c" ]`
	`b.delete("b"); puts b`
		`[3,5,1,2,4]`
		`["a", "c"]`

delete_at(1)	Видалити елемент з індексом 1. Змінює масив. `a = [6,3,5,1,2,4]; puts a.sort.reverse`	`{6,5,1,2,4]`

delete_if	Видаляє всі елементи масиву для яких значення всередині блоку дорівнює true. Змінює масив.
	`a = [ "a", "b", "c" ]`
	`a.delete_if {|x| x >= "b" } #=> ["a"]`
	`puts a #=> ["a"]`	

Таблиця 4.3.

Складні методи-ітератори

inject	inject – акумулює метод з двома параметрами, який від ітерації до ітерації зберігає певне значення. Класичний приклад використання методу inject — це знаходження суми елементів масиву чисел:
	`res = [1, 2, 3].inject{|sum, num| sum += num} #=> 6`
	Загальний вигляд метода-ітератора наступний:
	res = arr.inject (start) {|result, element| expression}
	start – початкове значення, що привласнюється змінної result. Наприклад, при необхідності накопичення суми дане значення дорівнює 0, а при накопиченні обробки воно дорівнює 1.
	result – змінна, в котру записується результат.
	element – т.з. «вікно», в якому проглядаються елементи починаючи з початкового в розглянутій послідовності (колекції) наявних елементів. На першому кроці `element = arr[0]`, на другому – `element = arr[1]` і т.д.
	expression – вираз, що пов'язує *result* и *element*. Для накопичення в змінній *result* добуток декількох элементів массву *expression* буде мати вигляд:
	`result * element`
	Для отримання добутку всіх елементів масиву a можна записати наступний ітератор:
	`a = [6,3,5,1,2,4];`
	`mul = a.inject(1){ |res, elem| res * elem}`
	`puts mul`
		6
		
		720

find_all	Метод find_all витягує послідовно з розглянутого масиву всі елементи, що задовольняють певній умові, і записує їх в новий масив. Загальний вигляд:
	`arr_res = arr.find_all { |element| condition }`
	element – поточний (що розглядається) елемент масива (колекції)
	condition – логічний вираз або кілька виразів, пов'язаних логічними операціями (`and, or, not`)
	Приклад застосування метода `find_all` для пошуку негативних єлементів масиву:
	`a = [6,-3,5,-1,2,-4];`
	`arr_neg = a.find_all{ |elem| (elem <0) }`
	`puts arr_neg #=> [-3, -1, -4]`
	`puts arr_neg.size #=> 3`
		3

4.5. Створення та заповнення багатовимірних масивів

Для Ruby двовимірний масив – це не більше ніж масив, що містить одновимірні масиви. Як наслідок, такі масиви не повинні бути прямокутними (*тобто число елементів в різних рядках може відрізнятися!*), що дозволяє значно економити пам'ять. Ось кілька прикладів двовимірних масивів:

[[1], [2, 3], [4]] # різна довжина елементів-масивів

[[1, 2], [3, 4]] # однакова довжина

[["Hi", "Hello"], ["By", "All the best"]] # двовимірний масив символьних даних (класика)

[["Hi ", "Hello"], [1, ["By", " All the best"]]] # гібрид двох-трьох-вимірного масиву

• двомірність масиву засобами мови не відслідковуються. Цілком можуть виникнути гібриди масивів різних розмірів.

• підмасиви всередині двовимірного масиву можуть мати довільну довжину.

• Елементи з двовимірного масиву виймаються послідовно: спочатку елемент-масив, потім елемент масиву.

Наприклад, двовимірний масив розміром 3x3 можна задати так

```
a = [
      [11, 12, 13],
      [21, 22, 23],
      [31, 32, 33]
    ]
```

Аналогічно можна створити масив будь-якої необхідної розмірності, хоча масиви розмірності, більшою, ніж два, зустрічаються досить рідко. До елементів таких масивів можна дістатися, послідовно вказуючи індекси необхідних елементів. Якщо потрібно тільки створити багатовимірний масив, не заповнюючи його даними, то можна використовувати наведені нижче можливості і конструкції мови Ruby.

4.6. Створення і заповнення двовимірних (n × m) масивів

Більшість математичних задач, особливо пов'язаних з матрицями, призводять до необхідності формування і заповнення двовимірних масивів.

Розглянемо кілька способів виконання таких завдань.

Перший спосіб:

```
# створення порожнього масиву a:
a = Array.new(3)
# створення 3 нових масивів(рядків) та переміщення їх у масив a
a[0] = Array.new(3)
a[1] = Array.new(3)
a[2] = Array.new(3)
a[1][1] = 123
p a  # [[nil, nil, nil], [nil, 123, nil], [nil, nil, nil]]
```

Другий спосіб.

Створення двох рядків безпосередньо в структурі масиву і заповнення їх елементами за допомогою методу `each_index`:

```
a = [Array.new(10), Array.new(10)]
p a # Виведення не заповненого масиву
a.each_index {|i|
  a[i].each_index{|j|
    a[i][j] = (i+2)*(j+3)
  }
}
puts a[0][0] #=> 6
puts a[0][1] #=> 8
puts a[1][0] #=> 9
puts a[1][1] #=> 12
p a # Виведення заповненого масиву
```

Третій спосіб. Послідовне використання методу-ітератора map. Здається розмірність двовимірного масиву, масив заданої розмірності будується і заповнюється випадковими числами за допомогою методу rand.

```
  m=4 ; n=4
#      Или таким образом с экрана
#  m = gets().to_i
#  n = gets().to_i

  b=Array.new(m).map!{Array.new(n)}

  for i in 0..3 do
  for j in 0..3 do

    if i == j then
         b[i][j] = 1
    else
         b [i][j] = rand(10)
    end

  end
  end

  p b
```

Результат роботи коду:

```
[[1, 6, 4, 2], [4, 1, 1, 5], [7, 8, 1, 7], [3, 6, 6, 1]]
```

Розглянемо створення масиву розміру 3x3, який повинен бути заповнений за наступною схемою: в першому рядку масиву - поточні рік, номер місяця і дня місяця, в другій - назва місяця, день і найменування дня тижня, в третьому рядку - поточні година, хвилина і секунда .

Запустіть код програми і отримаєте поточні значення: сьогоднішнього числа, місяця, дня тижня, а також часу запуску програми.

```
myArray = (0..2).map{ Array.new 3 }
mouth = %w(January February March April May June
   July August September October November December)

dayOfWeek = %w(Sunday Monday Tuesday Thursday
      Friday Saturday)
t = Time.now
myArray[0][0] = t.year
myArray[0][1] = t.month
myArray[0][2] = t.day
myArray[1][0] = t.month
myArray[1][1] = t.day
myArray[1][2] = t.wday
myArray[2][0] = t.hour
myArray[2][1] = t.min
myArray[2][2] = t.sec

print "Segodnia #{myArray[0][2]}.",
  "#{myArray[0][1]}.#{myArray[0][0]}\n"
print "Chislo:  #{myArray[1][1]}, ",
  "mesiats:  #{myArray[1][0]},",
  " den nedeli: #{myArray[1][2]}\n"
print "Seychas  #{myArray[2][0]} chas, ",
 "#{myArray[2][1]} min,", " #{myArray[2][2]}sec\n"
```

4.7. Сумування, добуток, і агрегація елементів за допомогою ітераторів

Дуже часто виникає задача знайти суму/добуток всіх елементів масиву. Для цих цілей традиційно використовується метод-ітератор .inject. Для демонстрації його роботи, давайте знайдемо суму елементів масиву array:

```
array = [1, 2, 3, 4, 5]
s = array.inject(0){ |result, elem| result + elem }
puts s    #=> 15
```

Розглянемо детально роботу даної конструкції. Почнемо з нуля в дужках "`(0)`". Його слід інтерпретувати як ініціалізацію значення для обнулення змінної `result = 0` перед початком роботи ітератора. Тобто це початкове значення змінної result (змінної проміжного результату).

Далі йде оголошення двох змінних | result, elem |. Перша з них (result) призначена для зберігання проміжного результату. Друга (elem) - є своєрідним «вікном», в якому послідовно «пробігають» елементи масиву або блоку від початку до кінця. У кожен момент часу це «вікно» містить поточний елемент масиву (або послідовності).

Після оголошення змінних | result, elem | далі описаний алгоритм роботи ітератора, тобто їх взаємодії. В даному випадку Ітератор наказано кожен елемент масиву додавати до проміжної сумі: result + elem.

Приклад коду для обчислення суми елементів масиву array з використанням інтерпретатора.

```
array = [1, 2, 3, 4, 5]
s =array.inject(0){ |result, elem| result = result + elem }
   puts s   #=> 15
```

4.8. Використання бібліотек mathn та matrix

В Ruby існує бібліотека mathn. Унікальність її полягає в тому, що вона істотно розширює можливості стандартних чисел, додаючи до них раціональні числа (клас Rational) і комплексні числа (клас Complex). Клас Rational дозволяє ділити числа без похибок і витягувати корінь з від'ємних чисел за допомогою класу Complex. Одночасно з цим дана бібліотека додає до обчислень матричне і векторне числення (для цього додатково необхідно підключити опції complex і matrix).

З існуючих методів роботи з матрицями найбільш важливі:

- обчислення визначника (.det),
- обчислення оберненої матриці (.inv),
- пошук мінору матриці (.minor),
- перетворення матриці в масив (.to_a),
- множення (*), додавання (+), віднімання (-) і ділення (/) матриць.

Наприклад, для того, щоб обчислити визначник матриці, слід виконати нижченаведений код, застосувавши метод знаходження детермінанта матриці .det.

```
require 'matrix'
p  Matrix[[1.0,  -2.0,  3.0],  [3.0,  4.0,  -5.0],  [2.0,  4.0,
1.0]].det  #=> 62.0
```

Найпростіший спосіб створити матрицю – це використовувати наступний метод:

```
require 'mathn'
Matrix[[1, -2, 3], [3, 4, -5], [2, 4, 1]]
```

Для виведення матриці в її звичайному вигляді можна використовувати наступний спосіб:

```
require 'mathn'
array = [[1, -2, 3], [3, 4, -5], [2, 4, 1]]
Matrix[*array]

m1 = Matrix[*array]
puts m1.to_a.map(&:inspect)
```

Результат роботи коду:

```
D:\6w>matr2
[1, -2, 3]
[3, 4, -5]
[2, 4, 1]
```

4.9. Обчислення добутку двох матриць, оформлення у вигляді методу `matrix_product`

Нижче наведена програма, для перемноження двох матриць (`A * B`) за допомогою методів і ітераторів. Вхідними параметрами є дві матриці з числовими значеннями, записані у вигляді двох масивів (arr1, arr2). Результат обчислюється і присвоюється ім'я методу matrix_product. У прикладі наведено звернення до програми двома прямокутними масивами a і b і двома матрицями c і d.

```
a = [
      [11, 12, 13],
      [21, 22, 23],
      [31, 32, 33]
    ]
b = [
      [11, 12, 13],
      [21, 22, 23],
      [31, 32, 33]
    ]
require 'mathn'
c = [Vector[11, 12, 13], Vector[21, 22, 23], Vector[31, 32,
33]]
d = [Vector[11, 12, 13], Vector[21, 22, 23], Vector[31, 32,
33]]
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

def matrix_product(m1, m2)
  raise "Nonconformant arguments" if m1[0].size != m2.size

  Array.new(m1.size) do |i|
    Array.new(m2[0].size) do |j|
     m2.size.times.inject(0) { |acc, k| acc + m1[i][k] * m2[k][j] }
    end
  end
end
p matrix_product(a, b)
p matrix_product(c, d)
```

Результат обчислень:

```
D:\6w>array1
[[776, 812, 848], [1406, 1472, 1538], [2036, 2132, 2228]]
[[776, 812, 848], [1406, 1472, 1538], [2036, 2132, 2228]]
```

4.10. Завдання до лабораторної роботи

Завдання для програмування береться з файлу додатку до Лабораторних робіт. Номер варіанта відповідає номеру комп'ютера, на якому працює студент. Звіт до даної лабораторної роботи повинен містити.

1. Назву роботи.
2. Тему, мету і завдання до роботи.
3. Вихідний код збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

Назва роботи: Одновимірні та багатовимірні масиви.

Тема, мета і завдання до роботи.

Тема: Використання одновимірних і багатовимірних масивів у вирішенні математичних задач.

Мета: Використання мовних конструкцій мови Ruby для написання і налагодження програм.

Завдання: Математичне обгрунтування завдань для програмування береться з файлу із завданнями до Лабораторної роботи № 4. Номер варіанту відповідає номеру комп'ютера, на якому працює студент.

Самі завдання див. нижче.

Масиви і вектори.

Напишіть програми відповідно до заданих умов. Там, де можливо, використовуйте ітератори. Введення даних в масив довільний (наприклад, можна використовувати метод rand).

1. Задано одновимірний масив A (12), що містить цілочисельні елементи. Знайти кількість груп повторних входжень елементів в даному масиві. Наприклад, в масиві A = [1, 2, 2, 3] - 1 група (2, 2).

2. Заданий масив дійсних чисел B (16). Знайти суму парних значень їх цілих частин, а також суму їх індексів.

3. Задані два цілочисельних масиву A (10) і B (10). Знайти їх перетин, тобто новий масив, що містить елементи, однакові для двох вищевказаних масивів.

4. Заданий цілочисельний масив C (12). Зрушити вміст масиву C вправо на одну клітинку, перенісши останній елемент в першу позицію масиву.

5. Дан цілочисельний масив A (13), що містить негативні, нульові і позитивні елементи. Побудувати новий масив B, який містить на початку все нульові елементи, потім все негативні, а потім все позитивні елементи.

6. Дан цілочисельний масив A (24). Знайти суму парних негативних і суму непарних позитивних значень.

7. Задані два цілочисельних масиву A (14) і C (14). Побудувати масив B, що містить елементи, загальні (однакові) для масивів A і B (тобто входять одночасно і в A і в C).

8. Задано цілочисельний масив C (16), що містить, в тому числі, і деякі однакові елементи. Побудувати масив D (16), в якому на місцях одноразово зустрічаються елементів масиву C заносяться ці елементи, а повторювані елементи масиву C заносяться в масив D у вигляді чисел повторення елементів, тобто - 1, 2, 3 ...

9. Задано масив A (15), що містить нульові, негативні і позитивні числа. Побудувати новий масив C (15), що містить індекси нульових, негативних, а потім позитивних значень елементів.

10. Задані цілочисельні масиви A (12) і B (12). Побудувати масив C (2 * 12), що містить спочатку парні елементи і непарні елементи масиву B, а потім непарні і парні елементи масиву A.

11. Заданий цілочисельний масив A (14). Побудувати масив C (14), що містить спочатку зворотну послідовність значень від середина масиву A (A (7), A (6), ..., A (1)), а потім зворотний послідовність значень з кінця масиву (A (14), A (13), ..., A (8).

12. Задані два речових масиву A (15) і B (15). Побудувати новий цілочисельний масив D (15), що містить спочатку суми парних цілих частин дійсних чисел масиву A з цілими частинами значень масиву B з однаковими індексами, а потім суми непарних цілих частин значень з масиву A з цілими частинами значень масиву B, що мають однакові індекси.

Матриці та вектори

Завдання 1: Розробити коди програм, які виконують базові операції з матрицями і дозволяють вирішити такі завдання:

1. Виконати множення матриці B на число.
2. Виконати додавання двох матриць A і B.
3. Виконати транспонування матриці A.
4. Виконати множення матриці A на матрицю B.
5. Знайти слід матриці B.
6. Знайти внутрішнє (скалярний) добуток векторів X і Y.
7. Знайти зовнішнє твір векторів X і Y.
8. Знайти норму вектора X.
9. Знайти твір вектор-стовпця X на матрицю A.
10. Знайти твір вектор-рядка Y на матрицю B.

Основні вимоги до програм:

1. Розмірність матриць A і B є сталою і дорівнює $n \times n = 8 \times 8$.
2. Всі діагональні елементи матриці A рівні 1.
3. Всі інші елементи матриць A і B визначаються шляхом звернення до методу rand (10) класу Numeric.
4. Розмірності векторів X і Y рівні $n = 8$. Елементи векторів обчислюються шляхом звернення до методу rand (10) класу Numeric.
4. Для роботи з матрицями бажано використовувати:

-Методи класів Array.

-ітератори і цикли.

«Рішення лінійних алгебраїчних рівнянь методом Гаусса»

Завдання 2: Розробити програму, яка обчислює значення невідомих в системах лінійних алгебраїчних рівнянь (СЛАР) методом виключення Гауса, виду:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1,$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2,$$
$$\cdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n.$$

Алгоритм рішення таких СЛАР наведено в Додатку 2.

Результат перевірити із застосуванням програми Excel (!!!).

Основні вимоги до програми:

1. Назва програми: Рішення систем лінійних алгебраїчних рівнянь методом Гаусса.

2. Розмірність СЛАР n × n змінюється: від 3 × 3 до 9 × 9. Тобто n = [3, 4, ..., 9].

3. Всі діагональні елементи матриці A рівні 2.

4. Всі інші елементи матриці коефіцієнтів системи рівнянь при невідомих x рівні k + 2, де k - номер комп'ютера.

5. Вектор правих частин b завжди постійний і містить значення (зверху вниз)

[1, 2, 3, ..., 8, 9].

Так, наприклад, при k = 5 і розмірності матриці A - 5 x 5, матриця коефіцієнтів A і вектор правих b частин будуть виглядати наступним чином:

$$A = \begin{pmatrix} 2 & 7 & 7 & 7 & 7 \\ 7 & 2 & 7 & 7 & 7 \\ 7 & 7 & 2 & 7 & 7 \\ 7 & 7 & 7 & 2 & 7 \\ 7 & 7 & 7 & 7 & 2 \end{pmatrix} \qquad b = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$$

Програма повинна:

1) Дозволяти вводити розмірність системи n (від 3 до 9).

2) Діагностувати значення розмірності n матриці A, що виходять за межі [3..9].

3) Заповнювати матрицю коефіцієнтів системи рівнянь A.

4) Заповнювати вектор правих частин b.

5) Обчислювати вектор невідомих x і виводити його на форму.

ПРИМІТКА: Нижче представлена програма, яка реалізує метод Гаусса мовою Object Pascal

Консольний додаток для перевірки рішення системи лінійних алгебраїчних рівнянь на мові Object Pascal

```
Program Project1;

{$APPTYPE CONSOLE}
uses
  SysUtils;
 var    rrr   :real;
 Const          {Рішення рівнянь методом виключення Гауса. }
  NStr = 3;     {  Порядок рівняння буде задано за допомогою     }
    NClm = 3;  {    змінних  NStr, NClm  ==========}
        E = 1.0;
Var
  A : Array[1..NStr, 1..NClm] of Real;
   B, X  : Array[1..Nstr] of Real;
     k, m, i, j, g, p, q : Integer; akk : real;
   { TODO -oUser -consoled Main : Insert code here }

Begin {Готуємо матрицю і праві частини}
  for  i:= 1 to NStr do {//} for j := 1 to NClm do  A[i,j] := 0.6;
      for i := 1 to NStr do {//}  for j := 1 to NClm do
          if i = j  then  A[i,j] := E;
            for i := 1 to NStr do  B[i] := E;
                                                {Починаємо рішення}
 for k := 1 to NStr-1 do
   begin   akk:= a[k,k];   { Ведучий елемент }
      for j := k to NClm do        begin
          A[k,j] := A[k,j] / akk;     end;
            B[k] := B[k] / akk ; {Перетворення початкового рядка}
       g := k + 1; {Номер стовпчика для нових правих частин}
   { Виключаємо з системи невідомі у (NStr-k) рівняннях}
              for i := g to NStr do
             begin { Хід по рядках }
                    for j := g to NClm do   {Хід по рядку}
                      A[i,j] := A[i,j] - A[i,k] * A[k,j];
                    B[i] := B[i] - A[i,k] * B[k];
             end{po I};
   end {po K }; {Зворотній хід}
      X[NStr] := B[NStr] / A[NStr,NClm];
        for k := NStr-1 DownTo 1 do
         begin
               X[k] := B[k];
                 for m := NStr-1 DownTo k do
                   X[k] := X[k] - A[k,m+1] * X[m+1];
        end {po K }; {Обчислені невідомі X}
  {Друк невідомих}
  WriteLn('Kolichestvo neizvestnyh:  ', NStr:3);
  for k := 1 to NStr do   Write(X[k]:9:5);
  ReadLn;
 read ( rrr); writeln(rrr);
  read ( rrr);
 readln;
  end.
```

ЛАБОРАТОРНА РОБОТА №5.

Методи та блоки програми

Тема: Рішення математичних задач з використанням методів та блоків.

Мета: Використання інтерпретаторів Ruby для налагодження програм. Реалізація закритих методів за допомогою вбудованих методів і методів-ітераторів.Теоретичні відомості:

Теоретичні відомості:

5.1. Вбудовані методи мови Ruby

Всі вбудовані методи мови Ruby визначені в своїх класах (див. рис. 2.2). Тому їх виклик здійснюється для відповідних об'єктів, які є реалізаціями класів.

У найзагальнішому випадку вираз виклику методу складається з чотирьох частин.

1) Довільного виразу, чиїм значенням є опрацьований об'єкт, щодо якого викликається метод. Цей вираз продовжується символом. або символами :: для відділення його від імені методу, яке йде далі. Вираз і роздільник необов'язкові; якщо вони опущені, метод викликається для поточного об'єкта – self.

2) Імені методу, який викликається. Це єдина обов'язкова частина виразу виклику методу.

3) Значний аргументів, переданих методом. Перелік аргументів може бути укладений в круглі дужки, але зазвичай їх застосування носить необов'язковий характер. Якщо передається більше одного аргументу, вони відокремлюються один від одного комами. Кількість і типи необхідних аргументів залежать від визначення методу. Деякі методи взагалі не вимагають аргументів.

4) Необов'язково блоку коду, обмеженого фігурними дужками або парою ключових слів do-end. Метод може викликати цей код, використовуючи ключове слово yield («термін, що позначає передачу (повернення) об'єктів в викликають функції або програми»). Така можливість зв'язування довільного коду з викликом методу є основою роботи наявних в Ruby потужних методів-ітераторів.

Значним вираження виклику методу є значення останнього обчисленого виразу в тілі цього методу. Коли Ruby-інтерпретатор має іменами методу і об'єкта, для якого він викликається, він знаходить відповідну ухвалу зазначеного методу, використовуючи процес, відомий як «пошук методу», або «дозвіл імені методу».

5.2. Методи - глобальні функції.

Найпростіше викликаються методи класу Kernel, що є глобальними функціями по відношенню до всіх інших класів. До них відносяться: print, printf, p, puts, .display і деякі інші. Глобальні функції, визначені в Kernel, як і будь-які інші методи, визначені на верхньому рівні, знаходяться поза будь-яких класів. Зазвичай глобальні функції визначені в якості закритих методів класу Object. Наведемо кілька прикладів виклику одного з цих методів, наприклад методу p:

```
p # => повертає значення nil
p 1 # => повертає значення 1
p 2 + 2 # => повертає значення 4 (тут два методу: p і +)
p 2> 2 # => повертає значення false (тут два методу: p і>)
```

У першому рядку метод p викликається без аргументу; у другому рядку його аргументом є значення об'єкта 1, а в останніх двох рядках аргументами методу p є значення, які повертаються виразами, записаними з застосуванням методів + і > до двох об'єктів 2 і 2.

5.3. Методи - оператори.

Дещо складніше виконується виклик методів-операторів. Оператор – це лексема мови Ruby, яка представляє операцію (таку як додавання або порівняння), виконувану над двома або більшою кількістю операндів. Операнди є виразами, і оператори дозволяють об'єднувати ці вирази-операнди в більш крупні вирази. Числовий літерал 2 і оператор +, наприклад, можуть бути об'єднані в вираз 2 + 2 (як це було зроблено вище). А в наступному виразі об'єднані числовий літерал 2, вираз виклику методу і вираз посилання на змінну з оператором множення і оператором «менше, ніж»:

```
limit = 7;
2 * Math.sqrt (2) <limit # => повертає значення true
```

Оператор на основі методу має два аргументи і викликається як метод щодо свого лівого операнда (чи свого єдиного операнда в разі використання унарних операторів).

5.4. Методи класу Numeric і Сінглтон-методи.

Стандартні математичні вбудовані методи (наприклад, abs, sin, tan та ін.), що застосовуються зазвичай до одного вхідного аргументу (параметру) викликаються в двох видах.

1) У вигляді методу з одним аргументом, розташованим перед ім'ям методу. Наприклад: (-7.9).abs. Детальніше такі методи обговорюються в описі класу Numeric.

2) У вигляді *сінглтон* (singleton – одіночного) методу, визначеного для одиничних об'єктів модуля Math з одним аргументом, розташованим після імені методу. Наприклад: Math.sin(0.543). Детальніше такі методи обговорюються в описі домішки (mixin) з ім'ям Math, яка містить методи обчислення найпростіших тригонометричних і трансцендентних функцій.

Нижче наведені приклади виклику вищеописаних методів.

```
12.abs # => 12
(-34.56) .abs # => 34.56
-34.56.abs # => 34.56
Math.sqrt (0.3 + 0.9 / 7) # => 0.6546536707079772
Math.log ((- 2.71) .abs) # => 0.9969486348916096
```

5.5. Методи-ітератори

Як вже зазначалося вище, найбільш складні конструкції представляють собою *методи-ітератори*. До їх складу крім виразів, аргументів і імені методу входить блок коду довільної структури. Структура блоку залежить від конкретного завдання, розв'язуваної програмістом. наприклад:

```
"Abc" .each_byte {| c | printf "% c \ n", c} # => a, b, c
[ 'Earth', 'Venus', 'Mars']. Each {| p | puts p} # => Earth,
Venus, Mars
```

Тут методи-ітератори `.each_byte` і `.each` застосовуються для обробки об'єктів рядків і масивів.

Таким чином, приходимо до висновку, що серед вбудованих методів спостерігається досить широке розмаїття і в їх видах, і в способах їх використання. В цілому, можна відзначити, що в `Ruby` є такі методи:

- звичайні методи;
- методи – глобальні функції;
- методи-оператори;
- Сінглтон-методи;
- методи-ітератори;
- закриті методи (як правило, ті, які пишуться програмістами).

Тому важливо представляти взаємозв'язок між ними і їх використанням.

5.6. Взаємозв'язок методів і їх використання

Оскільки мовне середовище `Ruby` підтримує процедурний і функціональний стилі програмування, реалізовані програмістами в цій мові методи при їх опису в різних джерелах називаються по-різному:

– методами-ітераторами;
– Сінглтон (`singleton`) методами;
– функціями;
– процедурами;
– подпрограммамі.

Те ж, стосується і блоків, які в різних джерелах називаються по-різному: *контейнерами*, *замиканнями* або *анонімними методами*.

Розглянемо докладніше, яке функціональне навантаження виконують різні програмні компоненти в структурі програмного коду.

Програма – це упорядкована послідовність команд, що підлягає обробці. Головної програми в мові `Ruby` як такої (функції `main`) не існує; тобто виконання коду відбувається зверху вниз по рядках виконуваної програми.

У більш складних програмах на початку тексту можуть розташовуватися численні визначення, за якими слідує (концептуально) головна програма. Але навіть в цьому випадку програма виконується зверху вниз, так як в `Ruby` все визначення виконуються.

Підпрограма – це частина програми, призначена для виконання певного завдання.

Процедура – це частина програми, яка виконує деяку чітко визначену операцію над даними, які визначаються параметрами (формальними і фактичними). Ця частина коду може бути викликана з будь-якої точки програми, і при кожному виклику можуть пересилатися різні параметри. Термін «процедура», взагалі кажучи, використовується в контексті мов високого рівня (процедурних мов). У мові асемблера зазвичай вживається слово «підпрограма».

Функція являє собою спеціальний вид підпрограм, що відрізняється тим, що повертає в точку виклику результат, який присвоюється імені функції. Тому функції можна використовувати у виразах, так як вони формують значення, які вставляються в ті точки програми, де зустрічаються імена функцій.

Слід зазначити, що з точки зору синтаксису застосування конструкцій в більшості мов програмування виклик процедури є виразом, а виклик функції - операндом у виразі. Розглянемо, наприклад, наступний фрагмент програми:

```
1   a = [ "a", "b", "c", "d"]
2   b = []
3   print " a1= ", a , "\n" ;    #=>  [ "a", "b", "c", "d"]
4   a.collect { |index| index = index+"!" }
5   print " a2= ", a , "\n" ;
6   b = a.collect { |index| index = index+"!" } * 2
7   print " b1= ", b, "\n" ;  #=> ["a!", "b!", "c!", "d!", "a!", "b!", "c!", "d!"]
8   a.collect! { |index| index = index+"!" }
9   print " a3= ", a , "\n"   #=> ["a!", "b!", "c!", "d!"]
10    a[0] = Math.cos(0.5) + Math.sin(0.5) ;
11  print " a4= ", a , "\n"   #=> [1.3570081004945758, "b!", "c!", "d!"]
12                                      print "\n"
13  c = print b
14                                      print "\n"
15  print " c2= ", c, "\n"
16                                      print "\n"
17  c = print b * 2
18  # c = print b + print a
```

Рис. 5.1. Приклад коду програми

Перший висновок за результатами роботи вищенаведеної програми полягає в тому, що мова `Ruby` допускає використання вбудованих методів і в «процедурному стилі» (рядки 3, 4, 5, 7, 8, 9, 11) і в «функціональному стилі», тобто як операндів виразів (рядки 6, 10, 13, 17 і 18).

Як і в інших мовах програмування, вбудовані функції, що реалізують обчислення значень синуса і косинуса, є операндами у виразі перевизначення першого елемента масиву `a` (рядок 10).

Крім цього, дана програма демонструє, що метод-итератор `collect` (також як і його модифікація `collect!`) Веде себе як справжня функція в виразі, де масиву `b` присвоюється подвоєний результат перетворення масиву `a` (рядок 6) та, разом з тим, допускає « процедурний »запис (рядки 4 і 8).

Найбільш цікавими представляються вирази в рядках 13, 17 і 18, де метод `print` використовується як *функція*. Хоча даний оператор мови повинен являти собою процедуру виведення даних на екран комп'ютера, вираз:

```
c = print b # => [ "a !!", "b !!", "c !!", "d !!", "a !!", "b !!", "c !!", "d !!"]
```

виводить на екран вміст масиву `b`.

Виявляється, що виконання присвоювання методу `print b` змінної `c` призводить до виводу потоку даних на екран комп'ютера (рядок 13). Більш того, такий ж вираз з подвоєним значенням результату, що повертається виразом print `b*2` направляє на екран рядок подвоєної довжини (останній рядок на рис. 5.1).

Разом з тим, сама змінна `c` дані не приймає (рядок з ідентифікатором `c2` на рис. 5.1), а рядок 18 (`c = print b + print a`) не виконується, з видачею попередження про синтаксичну помилку:

```
H:/99 r/t1.rb:12: syntax error, unexpected tIDENTIFIER, expecting keyword_do or
'{' or '('
```

Другий висновок полягає в тому, що всі вбудовані методи мови `Ruby` повертають результат, який може бути тим чи іншим чином використовуватися в подальших обчисленнях. Таким чином, об'єктно-орієнтований характер мови `Ruby` призводить до «зсуву» фокуса застосовності процедур в строну їх функціонального використання.

Зроблені висновки цілком узгоджуються з функціональною спрямованістю методів в об'єктно-орієнтованому програмуванні, де вони представляють операції над об'єктами, які визначаються як частина опису класу. У більшості мов програмування не будь-яка операція є методом, але в мові `Ruby` все методи є операціями. При цьому терміни «метод», «повідомлення» і «операція» зазвичай взаємозамінні. У деяких об'єктно-орієнтованих мовах методи існують самі по собі і можуть перевизначатися підкласами. У ряді інших мов метод не може бути перевизначений. Він служить як частина реалізації узагальнених або віртуальних функцій, які можна перевизначати в підкласах. У мові `Ruby` допускається перевизначення методів для базових класів.

Сінглтон метод – це метод, визначений в єдиному екземплярі, шляхом визначення для одиничного об'єкта за рахунок вказівки перед ім'ям методу імені об'єкта (виділень точкою), для якого він визначається. наприклад:

```
def  Math.square(x)   # Визначення методу класу для модуля Math x*x
end
```

Замикання (`closure`) являє собою процедуру або функцію, в тілі якої присутні посилання на змінні, оголошені поза тілом цієї функції і не в якості її параметрів (а в навколишньому коді). Кажучи іншою мовою, замикання – це процедура або функція, яка посилається на вільні змінні в своєму лексичному контексті. Замикання, так само як і екземпляр об'єкта, є способом уявлення функціональності і даних, пов'язаних і упакованих разом.

Разом з тим, замикання – це особливий вид функції, яка визначена в тілі іншої функції (в Ruby це зазвичай метод-ітератор) і створюється кожен раз під час її виконання. У запису це виглядає як функція, яка перебуває цілком в тілі іншої функції. Наприклад, поєднання методу-ітератора each в поєднанні з блоком виглядають так:

```
[1,2,3,4,5,6,7,8,9,10].each {|value| print "#{value} "}
  #=> будуть виведені значення 1 2 3 4 5 6 7 8 9 10
```

При цьому вкладена внутрішня функція (що розташовується в фігурних дужках) містить посилання на локальні змінні зовнішньої функції (в даному контексті – елементи масиву [1, 2, ..., 10]). Кожного разу при виконанні зовнішньої функції відбувається створення нового екземпляра внутрішньої функції, з новими посиланнями на змінні зовнішньої функції.

Ітератори – це методи, які приймають і виконують блоки або об'єкти класу Proc.

Блок – це просто фрагмент коду, який, зазвичай, приєднаний до методу, процедури (proc) або Lambda-функції (lambda). Важливо запам'ятати, що блок коду – це не об'єкт, а будь-який код, визначений всередині фігурних дужок {i}, або всередині контейнера do-end.

Можна сказати, що всі перераховані вище програмні конструкції об'єднують фрагменти коду для повторного або неодноразового використання. Узагальнено можна вважати, що в Ruby є дві великі групи таких конструкцій.

До першої групи входять блоки, а в другу - різновиди методів, модулі і класи. І ті й інші обмежують область видимості локальних змінних в реалізованій програмі. У найзагальнішому випадку, тобто у відсутності зазначених конструкцій, областю видимості локальної змінної є вся програма цілком (тобто весь текст програми).

Нижче наведені види опису блоків і більш складних синтаксичних конструкцій (методів, модулів і класів), які об'єднують вирази й оператори мови Ruby, що призначені для вирішення тих чи інших завдань. І ті й інші накладають певні обмеження на видимість оточуючих їх змінних.

```
do...end
• {...}
• loop{...}        #приклади блоків
• proc {...}
• lambda {...}

• def    .... end
• module .... end      #більш складні конструкції(методи,
• class  .... end      # модулі, класи)
```

5.7. Блоки та їх використання

Блоки не можуть використовуватися автономно; вони можуть бути лише припустимим продовженням виклику методу. При цьому блоки можна розміщувати після виклику методу; якщо метод не є ітератором і ніколи не викликає блок за допомогою ключового слова `yield`. У цьому випадку блок буде мовчазно проігнорований. Межами блоків служать фігурні дужки `{}` або ключові слова `do` і `end`. Відкриває фігурна дужка або ключове слово `do` повинні бути на тому ж самому рядку, що і виклик методу, інакше `Ruby` вважатиме закінчення рядка кінцем інструкції і викличе метод без блоку:

```
# Виведення чисел від 1 до 10
1.upto(10) {|x| puts x } # Виклик методу та блоку у фігурних дужках
                         # на одному і тому рядку
1.upto(10) do |x| # Межами блоку служать ключові слова do-end
   puts x
end
1.upto(10)          # Блок не задан
{|x| puts x } # Синтаксична помилка: блок не слідує за викликом
```

За загальноприйнятою угодою фігурні дужки використовуються, коли блок поміщається на одному рядку, а ключові слова `do` і `end` використовується, коли блок займає декілька рядків. Але цим угода не вичерпується. `Ruby`-парсер прив'язує фігурну дужку `{`строго до тієї лексеми, яка передує цьому символу. Якщо не вставити аргументи методу в круглі дужки і використовувати в якості обмежувача блоку фігурну дужку, то блок буде пов'язаний з останнім аргументом методу, а не з самим методом, що може не відповідати задуманому алгоритму.

За загальноприйнятою угодою параметри блоку записуються на тому самому рядку, на якій розташований виклик методу і відкривається фігурна дужка або ключове слово `do`, але з точки зору синтаксису дотримуватися цього правила не обов'язково.

Блоки визначають нову область видимості змінних. Змінні, створені всередині блоку, існують тільки в межах цього блоку і за його межами не визначені. Але потрібно мати на увазі, що локальні змінні методу доступні для будь-якого блоку всередині цього методу. Тому якщо в блоці присвоюється значення змінної, яка вже була визначена за його межами, нова, локальна змінна блоку не створюється, натомість нове значення присвоюється тій змінній, яка вже існувала Іноді саме це і потрібно.

```
total = 0
data.each {|x| total += x } # Підрахунок суми елементів масиву data
puts total                  # Виведення суми
```

У глобальному сенсі блок – це частина програми, яка при своєму створенні захоплює змінні навколишнього середовища. У мові Ruby існують два види блоків:

1) прості блоки (контейнери): do ... end, loop {...};

2) анонімні методи (контейнери, ітератори або замикання): {...}, proc {...}, lambda {...}.

Дані конструкції використовуються в якості, так званих «ітераторів», що завершують вбудовані методи мови. Як правило, блок розташовується безпосередньо після виклику методу. Слід зазначити, що блоки, першого і другого виду не завжди взаємозамінні.

Нижче наведено приклад неоднозначного сприйняття парсером Ruby операторних дужок блоків do end і {} в операторі циклу for..do і методі-ітераторі each_line.

```
for i in 1..10 do print "#{i} " end #=> виведення 1 2 3 4 5 6 7 8 9 10 в рядок
for i in 1..10 { print "#{i} " } #=> виведення помилки
"a\nb\nc\n".each_line { |k| print k } #=> виведення a b c у стовпчик
"a\nb\nc\n".each_line do |k| print k end #=> виведення a b c в стовпчик
```

Якщо блок багатостроковий доцільніше користуватися формою з do ... end або {...}.

Разом з тим, як зазначалося вище, в останніх версіях Ruby ці блоки не взаємозамінні. Тому, ,

Блок 1) не буде виконано і результат операції буде відповідним чином діагностований інтерпретатором.

```
puts (1..3).map do
   random_number = rand()
   "Hi - number is rundom!\n"+
   random_number.to_s
 end
```

1)

```
puts (1..3).map {
   random_number = rand()
   "Hi - number is rundom!\n"+
   random_number.to_s
 }
```

2)

Блок 2) буде виконаний успішно і виведе на екран наступний результат:

```
Hi - number is rundom!
0.68933042493557
Hi - number is rundom!
0.8989744502130956
Hi - number is rundom!
0.4679015329259447
```

Реалізовані програмістом методи також можуть використовуватися в ітераторах. Наприклад, наведене нижче поєднання «зовнішнього» методу `square` з методами `puts` і `map` дасть наступні результати :

```
def  square(x)    #
     x*x              #
 end

puts   (1..3).map   {|f|   p   =
square(f)
   "Hi  -  number  is  square!\n"  +
p.to s
```

```
def  square(x) #
     x*x       #
 end

puts   (1..3).map   {|f|   p   =
square(f);
   "Hi  -  number  is  square!+
#{p}"
```

```
Hi - number is square!
1
Hi - number is square!
4
Hi - number is square!
9
```

```
Hi - number is square!+ 1
Hi - number is square!+ 4
Hi - number is square!+ 9
```

5.8. Особливості ітераторів при використанні блоків

Хоча цикли `while`, `until` і `for` є основною частиною мови `Ruby`, все ж, написання циклів істотно спрощується при використанні методів-ітераторів. Найбільш використовуваними серед них є такі:

```
3.times { puts "Дякую!" } # Виведення подяки 3 рази
data.each {|x| puts x }    # виведення кожного елемента x, який міститься у data
[1.2.3].map {|x| x*x }    # Обчислення квадратів значень елементів масиву.
factorial = 1             # Обчислення факторіалу n
2.upto(n) {|x| factorial *= x }
```

Всі ці методи – `times`, `each`, `map` і `upto`, є ітераторами, і вони взаємодіють з блоком коду, який слідує за ними. За всім цим стоїть складна структура управління – `yield` (термін, що позначає передачу (повернення) функції або програми).

5.9. Числові ітератори

Ядро `Ruby API` надає декілька стандартних ітераторів. Наявний в класі `Kernel` метод `loop` поводиться як нескінченний цикл, що запускає пов'язаний з ним блок знову і знову, поки в блоці нічого очікувати виконано return, break або інша інструкція, що здійснює вихід з циклу.

У класі `Integer` визначені три часто використовуваних ітератора. Метод `upto` викликає пов'язаний з ним блок по одному разу для кожного цілого числа, що знаходиться між тим цілим числом, для якого він викликаний, і тим цілим числом, яке йому передано в якості аргументу. наприклад:

```
4.upto(6) {|x| print x}  # -> виведення "456"
```

Як видно з прикладу, `upto` передає кожне ціле число пов'язаному з ним блоку і включає сюди як початкову, так і кінцеву точку ітерації. Загалом `n.upto (m)` запускає свій блок `m-n + 1` раз.

Якщо потрібно виконати числову ітерацію, використовуючи числа з плаваючою точкою, можна скористатися більш складним методом `step`, що визначається в класі `Numeric`. Наприклад, наступний ітератор починає працювати зі значення `0` і проводить ітерації з кроком `0.1` до тих пір, поки не досягне значення `1`:

```
(0..1) .step (0.1) {| x | puts Math.sin (x)}
0.step (12, 3) {| x | print x, ""} # 0 3 6 9 12
```

5.10. Методи

Методи визначаються за допомогою ключових слів `def..end`. Завдяки тому, що вказівка класу-носія методу необов'язкова, на мові `Ruby` можна програмувати в функціональному стилі, не піклуючись про створення класу-«носія» для кожної групи методів.

У таблиці 5.1 представлені основні способи формування методів і їх викликів.

Таблиця 5.1.

Формування та використання методів (функцій)

Опис	Реалізація
Визначення функції з параметрами	`def f(p1, p2, ...) ... end`
Визначення функції без параметрів	`def f ... end`
Виклик функції	`f(a, b, ...) або f a, b, ...`
Виклик функції без параметрів	`f`

Приклад формування методу.

```
def sqr (x)       # Визначення методу по імені sqr
                  # з одним параметром x
    x**0.5       # Повернення квадратного кореня із значення x
end                # Завершення методу

puts sqr(2)      #=> виведення 1.4142135623730951
```

Коли метод, подібний цьому визначено за межами класу або модуля, він фактично є глобальною функцією, а не методом, що викликається для об'єкта. Але з технічної точки зору такий метод стає закритим методом класу `Object`.

Метод також може бути визначений для одиничного об'єкта за рахунок вказівки перед ім'ям методу імені об'єкта, для якого він визначається (в даному контексті – `Math`). Подібні методи відомі як Сінглтон (`singleton`) методи (тобто методи, визначені в одиничному екземплярі), і саме так в `Ruby` визначаються методи класу:

```
def Math.square(x)
    x*x
end
```

5.11. Обчислення визначеного інтеграла

Для наближеного обчислення певного інтеграла є перевірена часом техніка. Наведений нижче метод integrate приймає початкове і кінцеве значення залежної змінної, а також збільшення. Четвертий параметр (який насправді не є параметром) - це блок. У блоці має обчислюватися значення функції від переданої в нього залежною змінною (тут слово «змінна» вживається в математичному, а не програмістські сенсі). Необов'язково окремо визначати функцію, яка викликається в блоці, але в даній реалізації це виконано.

```
def integrate(x0, x1, dx=(x1-x0)/1000.0)
  x = x0
    sum = 0
       loop do
        y = yield(x)
        sum += dx * y
        x += dx
        break if x > x1
 end
        sum
end

def f(x)
 x**2
end

z = integrate(0.0,5.0) {|x| f(x) }
puts z, "\n" # 41.72918749999876
```

Тут враховувався той факт, що блок повертає значення, яке може бути отримано за допомогою ключового слова `yield`. Також в програмі зроблені деякі припущення. По-перше, передбачається, що `x0` менше `x1` (в іншому випадку може вийти нескінченний цикл). По-друге, передбачається, що функцію можна обчислити в будь-якій точці заданої області.

5.12. Обчислення дисперсії і стандартного відхилення

Дисперсія – це міра «розкиду» значень з набору. В даному випадку не розрізняються зміщені і незміщені оцінки. Стандартне відхилення, яке зазвичай позначається буквою σ, дорівнює квадратному кореню з дисперсії.

```
data = [2, 3, 2, 2, 3, 4, 5, 5, 4, 3, 4, 1, 2]

  def variance(x)
    n=x.size
   m =  x.inject(0){ |res, elem| res= res + elem}
   m=  m.fdiv n
sum = 0.0
        x.each {|v| sum += (v-m)**2 }
        sum/x.size
  end

def sigma(x)
        Math.sqrt(variance(x))
end

puts variance(data) # 1.4556213017751476
puts sigma(data)    # 1.2064913185660093
```

Слід зазначити, що у функції `variance` викликаються методи `.size`, `.inject` та `.each`.

5.13. Завдання до лабораторної роботи

Завдання для програмування береться з файлу додатку до Лабораторних робіт. Номер варіанта відповідає номеру комп'ютера, на якому працює студент. Звіт до даної лабораторної роботи повинен містити.

1. Назву роботи.
2. Тему, мету і завдання до роботи.
3. Вихідний код збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

ЛАБОРАТОРНА РОБОТА №6.

Основи ООП в Ruby

Тема: Рішення математичних задач з використанням об'єктно орієнтоване програмування.

Мета: Вивчення основних принципів ООП на мові Ruby та їх застовуання у створення програм.

Теоретичні відомості:

У мові Ruby є всі елементи, які прийнято асоціювати з об'єктно-орієнтованими мовами: об'єкти з інкапсуляцією і приховуванням даних, методи з поліморфізмом і перевизначенням, класи з ієрархією і спадкуванням. Але Ruby йде ще далі, додаючи обмежені можливості створення метаклассом, синглетні методи, модулі і класи-домішки. Схожі ідеї, тільки під іншими іменами, зустрічаються і в інших об’єктно-орієнтованих мовах, але одна і та ж назва може приховувати тонкі відмінності.

6.1. Об’єкти

В Ruby всі числа, рядки, масиви, регулярні вирази і багато інших сутностей фактично є об’єктами. Робота програми полягає у виклику методів різних об’єктів:

```
3.succ                  #       4
«abc».upcase            #       «ABC»
[2,1,5,3,4].sort        #       [1,2,3,4,5]
someObject.someMethod   #       результат
```

В Ruby кожен об’єкт є екземпляром якогось класу. Клас містить реалізацію методів. Крім інкапсуляції власних атрибутів і операцій об’єкт в Ruby має унікальний ідентифікатор.

6.2. Вбудовані класи

Останнім часом вже понад 30 класів вбудовано в Ruby. Як і в багатьох інших об'єктно-орієнтованих мовах, в ньому не допускається множинне спадкування. Сучасні мови часто побудовані відповідно до моделі одинарного наслідування. Ruby підтримує модулі і класи-домішки. Також реалізовані ідентифікатори об'єктів, що дозволяє будувати стійкі, розподілені і переміщувані об'єкти.

Для створення об'єкта існуючого класу зазвичай використовується метод new:

```
myFile = File.new("textfile.txt","w")
myString = String.new("Строковий об'єкт")
```

Посилання на об'єкти зберігаються в змінних. Вище вже зазначалося, що самі змінні не мають типу і не є об'єктами - вони лише посилаються на об'єкти.

```
x = "abc"
```

З цього правила є виняток: невеликі незмінні об'єкти деяких вбудованих класів, наприклад Fixnum, безпосередньо копіюються в змінні, які на них посилаються. Розмір цих об'єктів не перевищує розміру покажчика, тому зберігати їх таким чином більш ефективно. У такому випадку під час привласнення створюється копія об'єкта, а купа не використовується. При привласненні змінних посилання на об'єкти усуспільнюється.

6.3. Модулі та класи-домішки.

Більшість вбудованих методів успадковуються від класів-предків. Варто відзначити методи модуля Kernel, які домішуються до суперкласу Object. Оскільки клас Object повсюдно доступний, то і додані до нього з Kernel методи також доступні в будь-якій точці програми. Ці методи грають важливу роль в Ruby.

Терміни «модуль» і «домішка» – майже синоніми. Модуль являє собою набір методів і констант, зовнішніх по відношенню до програми на Ruby. Його можна використовувати просто для управління простором імен, але основне застосування модулів пов'язано з «підмішуванням» його можливостей в клас (за допомогою директиви include). В такому випадку він використовується як клас-домішка. Цей термін запозичений з мови Python. Варто відзначити, що в деяких варіантах LISP такий механізм існує вже більше двадцяти років.

Модуль в Ruby – це не зовнішній вихідний текст і не двійковий файл (хоча може зберігатися і в тому, і в іншому вигляді). Це об'єктно-орієнтована абстракція, в чомусь схожа з класом. Прикладом використання модуля для управління простором імен служить модуль Math. Так, щоб отримати визначення числа π, необов'язково включати модуль Math за допомогою пропозиції include; досить просто написати Math::PI.

Домішка дає спосіб отримати переваги множинного спадкоємства, не обтяжені характерними для нього проблемами. Можна вважати, що це обмежена форма множинного спадкоємства, але творець мови Мац називає його одиночним успадкуванням з поділом реалізації.

Include включає імена із зазначеного простору імен (модуля) в поточне. Метод extend додає об'єкту функції з модуля. У разі застосування include методи модуля стають доступні як методи екземпляра, а в разі extend – як методи класу.

Операції load і require не мають нічого спільного з модулями: вони відносяться до вихідних і двійкових файлів (завантажуваних динамічно або статично). Операція load читає файл і вставляє його в поточну точку початкового тексту, так що починаючи з цієї точки стають видимі все визначення, що знаходяться в зовнішньому файлі. Операція require аналогічна load, але не завантажує файл, якщо він вже був завантажений раніше.

Програмісти, які тільки почали освоювати Ruby, особливо які мають досвід роботи з мовою C, можуть спочатку плутати операції require і include, які ніяк не пов'язані між собою.

6.4. Створення класів

В Ruby є безліч вбудованих класів, але також є можливість визначати нові. Для визначення нового класу застосовується така конструкція:

```
class ClassName
# ...
end
```

Саме *ім'я класу* – це глобальна константа, тому воно повинно починатися з великої літери. Визначення класу може містити константи, змінні класу, методи класу, змінні екземпляра і методи екземпляра. Дані рівня класу доступні всім об'єктам цього класу, тоді як дані рівня екземпляра доступні тільки одному об'єкту

Класи в Ruby не мають імен. «Ім'я» класу – це лише константа, яка посилається на об'єкт типу Class (оскільки в Ruby Class – це клас). На один і той самий клас можуть посилатися кілька констант, їх можна привласнювати змінним, як будь-які інші об'єкти (оскільки в Ruby Class – це об'єкт). Якщо ви трохи заплуталися, не турбуйтеся.

Наведемо визначення простого класу:

```
class Friend
 @@myname = "Ендрю"           # змінна класу

 def initialize(name, sex, phone)
  @name, @sex, @phone = name, sex, phone
  # змінні екземпляра
 end

 def hello # метод екземпляра
  puts "Привіт, я #{@name}."
 end

 def Friend.our_common_friend # метод класу
  puts " Ми всі друзі #{@@myname}."
 end
end

f1 = Friend.new("Сюзанна","F","555-0123")
f2 = Friend.new("Том","M","555-4567")

f1.hello                      # Привіт, я Сюзанна.
f2.hello                      # Привіт, я Том.
Friend.our_common_friend      # Ми всі друзі Ендрю.
```

Оскільки дані рівня класу доступні в усьому класі, їх можна форматувати в момент визначення класу. Якщо визначено метод з ім'ям initialize, то гарантується, що він буде викликаний відразу після виділення пам'яті для об'єкта.

Цей метод схожий на традиційний конструктор, але не виконує виділення пам'яті. Пам'ять виділяється методом new, а звільняється складальником сміття.

Для управління видимістю методів класу можна користуватися модифікаторами `private`, `protected` і `public`. При цьому, змінні екземпляру завжди закриті, звертатися до них ззовні класу можна тільки за допомогою методів доступу. Кожен модифікатор приймає як параметр символ, наприклад: `foo`, а якщо він опущений, то дія модифікатора поширюється на всі наступні визначення в класі. приклад:

```
class MyClass
 def method1
  # ...
 end
 def method2
  # ...
 end

 def method3
  # ...
 end
 private :method1
 public
:method2
 protected :method3
private
 def my_method
  # ...
 end
 def another_method
  # ...
 end
end
```

В цьому класі метод `method1` закритий, `method2` відкритий, а `method3` захищений. Оскільки далі викликається метод `private` без параметрів, то методи `my_method` і `another_method` будуть закритими.

Рівень доступу `public` не потребує пояснень, він не накладає ніяких обмежень ні на доступ до методу, ні на його видимість.

Рівень `private` означає, що метод доступний виключно всередині класу або його підкласів і може викликатися тільки в «функціональній формі» від імені `self`, причому викликає об'єкт може вказуватися явно чи розумітися неявно.

Рівень `protected` означає, що метод викликається тільки всередині класу, але, на відміну від закритого методу, не обов'язково від імені `self`.

За замовчуванням всі певні в класі способи відкриті. Виняток становить лише `initialize`.

Методи, визначені на верхньому рівні програми, теж за замовчуванням відкриті. Якщо вони оголошені закритими, то можуть викликатися тільки в функціональній формі (як, наприклад, методи, визначені в класі `Object`). Класи в `Ruby` є об'єктами – екземплярами метакласу `Class`. Класи завжди конкретні, абстрактних класів не існує. Однак теоретично можна реалізувати і абстрактні класи.

6.5. Методи та атрибути

Методи зазвичай використовуються в поєднанні з простими екземплярами класів і змінними, причому, об'єкт, який викликається синтаксично, відділяється від імені методу точкою (`receiver.method`). Якщо ім'я методу є знаком пунктуації, то точка опускається. У методів можуть бути аргументи, наприклад:

```
Time.mktime (2000, "Aug", 24, 16, 0)
```

Оскільки кожен вираз повертає значення, то виклики методів можуть зчіплюватися:

```
3.succ.to_s
/(x.z).*?(x.z).*?/.match("x1z_1a3_x2z_1b3_").to_a[1..3]
3 + 2.succ
```

Деяким методам можна передавати блоки. Це вірно для всіх ітераторів – як вбудованих, так і визначених користувачем. Блок зазвичай вкладається в операторні дужки `do..end` або в фігурні дужки `{}`. Але він не розглядається так само, як попередні параметри, якщо такі існують. Приклад виклику методу `File.open`:

```
my_array.each do |x|
     some_action
end
File.open(filename) { |f| some_action }
```

В `Ruby` є можливість визначати методи на рівні об'єкта (а не класу). Такі методи називаються синглетними; вони належать одному-єдиному об'єкту і не впливають ні на клас, ні на його суперкласу. Така можливість може бути корисна, наприклад, при розробці графічних інтерфейсів користувача: щоб визначити дію кнопки, задається синглетний метод для даної і тільки даної кнопки.

6.6. Кодування під час виконання.

Вище вже були згадані директиви `load` і `require`. Це не вбудовані пропозиції і не керуючі конструкції, а методи. Тому їх можна викликати, передаючи змінні або вирази як параметри, в тому числі умовно. Код можна будувати і інтерпретувати по частинах.

Розглянемо наведений нижче метод calculate і викликає його код:

```
def calculate(op1, operator, op2)
 string = op1.to_s + operator + op2.to_s
 # Передбачується, що operator – це рядок;
 # побудуємо рядок, який складається з оператора та операндів.
 eval(string)  # Обчислення та повернення значення.
end

@alpha = 25
@beta = 12

puts calculate(2, "+",2)          # Виводиться 4
puts calculate(5, "*", "@alpha") # Виводиться 125
puts calculate("@beta", "**", 3) # Виводиться 1728
```

Ще один приклад. Програма запитує у користувача ім'я методу і один рядок коду.

Потім цей метод визначається і викликається: `puts "Ім’я методу "`

```
meth_name = gets
puts "Рядок коду: "
code = gets

string = %[def #{meth_name}\n #{code}\n end] # Будування рядку.
eval(string) # Визначення методу.
eval(meth_name) # Виклик методу.
```

Найчастіше, для рішення задачі, необхідно написати програму, яка могла б працювати на різних платформах або при різних умовах, але при цьому зберегти загальний набір вихідних текстів. Для цього в мові С застосовуються директиви `#ifdef`, але в `Ruby` всі визначення виконуються. Не існує такого поняття, як «етап компіляції»; всі конструкції динамічні, а не статичні. Тому для прийняття рішення такого роду можна обчислити умову під час виконання.

6.7. Наслідування

Для прикладу розглянемо наступний код, він буде складатися з двох класів: Monkey і Human, де Human, згідно революційної еволюційної теорії Чарльза Дарвіна успадковується від класу Monkey:

```
1   class Monkey
2   def initialize(height, weight)
3   @height = height
4   @weight = weight
5   end
6
7   def height
8   @height
9   end
10
11  def weight
12  @weight
13  end
14
15  def height=(height)
16  @height = height
17  end
18
19  def weight=(weight)
20  @weight = weight
21  end
22  end
23
24  class Human < Monkey
25  def initialize(height, weight, name)
26  super(height, weight)
27  @name = name
28  end
29
30  def name
31  @name
32  end
33
34  def name=(name)
35  @name = name
36  end
37  end
```

Працюючи з абстрактними типами даних, необхідно явно визначати об'єкт за допомогою методу .new класу:

```
39  m = Monkey.new(120,55)
40  h = Human.new(180, 90, 'Ivan')
```

Спадкування надає можливість повторного використання коду, та позбавляє програміста від повторень в коді. Маючи клас Human, можна успадковувати від нього класи WhiteHuman і BlackHuman, які будуть мати абсолютно однакові властивості, за винятком кольору шкіри і форми носа. Замість написання двох нових класів, всі властивості та методи успадковуються від базового класу - Human доповнюючи його новими властивостями і методами, або переписуючи старі.

```
1   class BlackHuman < Human
2
3   class << self
4   def obj_count
5   count = 0
6   ObjectSpace.each_object(self){|obj| count += 1}
7   end
8   end
9
10  def initialize(height, weight, name)
11  super(height, weight, name)
12  @skin_color = :black
13  end
14
15  def skin_color
16  @skin_color
17  end
18
19  def skin_color=(skin_color)
20  @skin_color = skin_color
21  end
22  end
23
24  bh = BlackHuman.new(180,90,"John")
25  bh2 = BlackHuman.new(185,90,"Robert")
26  bh3 = BlackHuman.new(170,70,"Thomas")
27
28  puts BlackHuman.obj_count # 3
```

Як видно з прикладу, виклик методів класу не відрізняється від виклику методів об'єкта, єдина відмінність - це приймач методу.

6.8. Завдання до лабораторної роботи

Завдання для програмування береться з файлу додатку до Лабораторних робіт. Номер варіанта відповідає номеру комп'ютера, на якому працює студент. Звіт до даної лабораторної роботи повинен містити.

1. Назву роботи.
2. Тему, мету і завдання до роботи.
3. Вихідний код збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

ЛАБОРАТОРНА РОБОТА №7.

Багатопоточність в Ruby

Тема: Основи багатопоточності у мові Ruby.

Мета: Вивчення основ багатопоточності у мові Ruby. Отримання навичок з використання потоків.

Теоретичні відомості:

Найчастіше найбільш простим способом виконати одночасно дві речі є використання потоків в Ruby. Вони виконуються всередині процесів, вбудованих в інтерпретатор Ruby. Це дозволяє повністю переносити потоки Ruby, незалежно від операційної системи

7.1. Створення потоків Ruby

Створення нового потоку досить прямолінійно. Наступний код – простий приклад. Він паралельно викачує набір Веб-сторінок. Для кожного запитуваної для закачування URL код створює окремий потік, який управляє HTTP-транзакцією.

```
require 'net/http'
pages = %w( www.rubycentral.com slashdot.org www.google.com )
threads = []
for page_to_fetch in pages
    threads << Thread.new(page_to_fetch) do |url|
        h = Net::HTTP.new(url, 80)
        puts "Fetching: #{url}"
        resp = h.get('/', nil )
        puts "Got #{url}: #{resp.message}"
    end
end
threads.each {|thr| thr.join }
```

Результат роботи коду:

```
Fetching: www.rubycentral.com
Fetching: slashdot.org
Fetching: www.google.com
Got www.google.com: OK
Got www.rubycentral.com: OK
Got slashdot.org: OK
```

Нові потоки створюються викликом `Thread.new`. Це задає блок, що містить код, який буде виконуватися в новому потоці. У прикладі блок використовує бібліотеку `net/http` для вилучення головної сторінки зазначених сайтів. Трасування явно показує, що ці вилучення виконуються паралельно.

При створенні потоку, вказується необхідний URL в якості параметра. Цей параметр передається в блок у вигляді змінної url. Потік має загальний доступ до всіх глобальних змінних, змінним примірника і локальних змінних, які є на момент запуску потоку.

В цьому випадку, всі три потоки будуть ділитися змінною page_to_fetch. Коли запускається перший потік, page_to_fetch приймає значення «www.rubycentral.com». А між тим, цикл, що створює потоки, до сих пір працює. В наступний момент часу page_to_fetch встановлюється в «slashdot.org». Якщо перший потік ще не закінчив використовувати змінну page_to_fetch, то він несподівано почне використовувати її нове значення. Цей вид помилки дуже важко відстежити.

Однак, локальні змінні, створювані всередині блоку потоку, є дійсно локальними по відношенню до цього потоку - кожен потік буде мати свою копію адреси сторінки. Можна вказати будь-яке число аргументів в блоці за допомогою Thread.new.

7.2. Управління потоками

Коли програма Ruby завершується, всі потоки закінчуються, незважаючи на їх стан. Але є можливість почекати завершення окремого потоку шляхом виклику методу Thread#join. Виконуваний потік заблокується до того моменту, як поточний потік не завершиться. Якщо викликати join для кожного потока, то всі три запити будуть виконані перед завершенням основної програми. Якщо не потрібно блокувати потік назавжди, то можна передати в join параметр ліміту часу - якщо цей ліміт закінчиться перед завершенням потоку, виклик join поверне значення nil. Інший варіант join - метод Thread#value, який повертає значення останньої операції, виконаної в потоці.

Крім join, для управління потоками використовується кілька інших зручних операцій. Доступ до поточного потока можна завжди отримати, використовуючи Thread.current. Щоб отримати список всіх потоків, використовуть Thread.list, який повертає список всіх об'єктів Thread: і працюючих, і зупинених. Для визначення статусу окремого потоку використовують Thread # status і Thread # alive ?.

Додатково можна налаштувати пріоритет потоку, використовуючи Thread#priority =. Потоки з більшим пріоритетом будуть запускатися перед потоками з меншим пріоритетом.

7.3. Змінні потоку

Потік має звичайний доступ до всіх змінних, що знаходяться в області видимості під час його запуску. Локальні змінні блоку, що міститять код потоку, є локальними по відношенню до самого потоку і не мають до себе загального доступу.

Характерною рисою класу Thread є спеціальна можливість, що дозволяє створювати і мати доступ на ім'я до локальних змінних потоку. До об'єкту потоку можно відноситись як до хешу, встановлюючи значення елементів за допомогою [] = i читаючи їх за допомогою [].

У наступному прикладі кожен потік записує поточне значення лічильника в локальну змінну потоку з ключем mycount. Для здійснення цього код використовує рядок «mycount» в якості індексу об'єкта потоку.

```
count = 0
threads = []
10.times do |i|
    threads[i] = Thread.new do
        sleep(rand(0.1))
        Thread.current["mycount"] = count
        count += 1
    end
end
threads.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

Результат:

```
4, 1, 0, 8, 7, 9, 5, 6, 3, 2, count = 10
```

Головний потік чекає, поки завершаться інші потоки, а потім виводить значення лічильника, зафіксовані кожним потоком. Для ночності була додана випадкова затримка у кожному потоку перед записом значення лічильника.

7.4. Потоки і виключення

Якщо в потоці виникне необроблений виняток, зміниться значення прапора `abort_on_exception` та значення прапора налагодження інтерпретатора.

Якщо `abort_on_exception = false` і прапор налагодження не включений (стандартний стан), то необроблений виняток просто завершить поточний потік, а всі інші продовжать свою роботу. На практиці виключення ніяк візуально не виявити, поки для потоку, який викинув це виняток, що не буде викликаний join.

У наступному прикладі потік 2 роздувається і не може нічого вивести. Однак інші потоки продовжували свою роботу та залишили «сліди» з даних.

```
threads = []
4.times do |number|
    threads << Thread.new(number) do |i|
        raise "Error!" if i == 2
        print "#{i}\n"
    end
end
threads.each {|t| t.join }
```

Результат:

```
0
1
3
prog.rb:4: Error! (RuntimeError)
from prog.rb:8:in `join'
from prog.rb:8
from prog.rb:8:in `each'
from prog.rb:8
```

Є можливість перехопити виняток під час виконання `join`. Для цього до коду, який ми розглядали у попередньому прикладі слід добавити наступні рядки.

```
 Begin
        t.join
        rescue RuntimeError => e
        puts "Failed: #{e.message}"
    end
end
```

Результат:

```
0
1
3
Failed: Error!
```

Всередині циклу краще використовувати print для виведення числа, ніж puts, тому що puts розбиває свою роботу на дві складові: виводить свій аргумент, а потім виводить символ нового рядка. Між ними може запуститися потік, і висновок буде чергуватися. Викликаючи print одного рядка, який вже містить символ нового рядка, ми можемо обійти цю проблему.

7.5. Управління планувальником потоків

У добре спроектованому додатку потокам будуть просто робити свою роботу. Створення тимчасових залежностей в багатопотоковому додатку, як правило вважається поганим тоном, тому що це робить код набагато важче для сприйняття, а також унеможливлює оптимізацію виконання програми планувальником потоків.

Однак, іноді буде необхідно управляти потоками наявно. Наприклад, музичний автомат, який показує світломузику. Він повинен зупинитись у той час, коли зупиниться музика. Можна використовувати два потоки у вигляді схеми виробник-споживач, де споживач повинен почекати, якщо виробник має незавершені замовлення.

Клас `Thread` надає набір методів для управління планувальником потоків. Виклик `Thread.stop` зупиняє поточний потік, а виклик `Thread#run` запускає окремий потік. `Thread.pass` запускає планувальник, щоб доручити виконувати іншому потоку, а `Thread#join` і `Thread#value` припиняють викликає потік, поки задані потоки на завершаться.

Цю особливість можна продемонструвати у наступній програмі. Вона створює два дочірніх потоки: t1 і t2, кожен з яких є екзмепляром класу `Chaser`. Метод `chase` інкрементує лічильник, але не дає йому стати більшим, ніж на два порівняно з лічильником в іншому потоці. Для зупинки цього збільшення метод викликає `Thread.pass`, які дозволяє запуститися методу `chase` в іншому потоці. Відразу після старту потоки припиняються, а потім у випадковому порядку запускаються.

```
class Chaser
    attr_reader :count
    def initialize(name)
        @name = name
        @count = 0
    End
    def chase(other)
        while @count < 5
            while @count - other.count > 1
                Thread.pass
            end
            @count += 1
            print "#@name: #{count}\n"
        end
    end
end

c1 = Chaser.new("A")
c2 = Chaser.new("B")
threads = [
    Thread.new { Thread.stop; c1.chase(c2) },
    Thread.new { Thread.stop; c2.chase(c1) }
]
start_index = rand(2)
threads[start_index].run
threads[1 - start_index].run
threads.each {|t| t.join }
```

Результат роботи програми:

```
B: 1
B: 2
A: 1
B: 3
A: 2
B: 4
A: 3
B: 5
A: 4
A: 5
```

7.6. Завдання до лабораторної роботи

Завдання для програмування береться з файлу додатку до Лабораторних робіт. Номер варіанта відповідає номеру комп'ютера, на якому працює студент. Звіт до даної лабораторної роботи повинен містити.

1. Назву роботи.
2. Тему, мету і завдання до роботи.
3. Вихідний код збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

ЛАБОРАТОРНА РОБОТА №8.

Використання потоків в Ruby. Паралельне та розподілене програмування.

Тема: Використання потоків для математичних обчислень у мові Ruby.

Мета: Вивчення способів використання потоків у мові Ruby. Отримання навичок з паралельного та розподіленого програмування.

Теоретичні відомості:

8.1. Потоки і обчислювальні операції

Програмісти на різних мовах програмування можуть поліпшувати швидкодію програм за допомогою потоків. Ruby не э виключенням, для цього нижче приведений наступній код, у якому 10 000 разів по два рази обчислюється факторіал 1000 послідовно, а в іншому паралельно.

```
require 'benchmark'
     factorial = Proc.new { |n|
     1.upto(n).inject(1) { |i, n| i * n }
     }
     Benchmark.bm do |x|
          x.report('sequential') do
               10_000.times do
                    2.times do
                         factorial.call 1000
                    end
               end
          end
x.report('thready')do
     10_000.times do
          threads = []
          2.times do
               threads << Thread.new do
                    factorial.call 1000
               end
          end
          threads.each &:join
          end
     end
end
```

Результат виявився досить несподіваним (використовуючи Ruby 2.0) – паралельне виконання завершилось на секунду довше розподіленого.

```
                user     system      total        real
sequential 24.130000   1.510000  25.640000 (25.696196)
thready    24.600000   2.420000  27.020000 (26.877708)
```

Одна з причин - це ускладнений код плануванням потоків, а друга – Ruby в один момент часу використовував тільки одне ядро для виконання цієї програми. На жаль, можливості змусити Ruby використовувати кілька ядер для одного Ruby-процесу на даний момент неможливо.

Вибираючи кількість одночасних потоків необхідно пам'ятати – без втрати продуктивності можливо розташовувати безліч потоків, що займаються IO операціями на одному ядрі. Але будуть невеликі втрати в продуктивності, якщо кількість потоків буде перевершувати кількість ядер у випадку з обчислювальними операціями.

У випадку з оригінальною імплементацією Ruby (MRI) рекомендується використовувати тільки один потік для обчислювальних операцій. Справжнього паралелізму за допомогою потоків ми можемо домогтися тільки використовуючи jRuby і Rubinius.

8.2. Паралелізм на рівні процесів

Як було сказано вище, Ruby MRI для одного процесу Ruby (в Unix системах) може використовувати ресурси тільки одного ядра в одним момент часу. Одним з варіантів, як можна обійти цей недолік, є використання Fork(форк) процесів, наприклад:

```
read, write = IO.pipe
result = 5 pid = fork do
     result = result + 51
     Marshal.dump(result, write)
     exit 0
     end
     write.close
     result = read.read
     Process.wait(pid)
     puts Marshal.load(result)
```

Форк процесу, на момент створення, копіює значення змінної result, що дорівнює 5, але головний процес не побачить подальшого перетворення змінної всередині форку, тому було необхідно налагодити сполучення між форком і головним процесом за допомогою IO.pipe.

Такий метод є дієвим, але досить громіздким і незручним. За допомогою модуля DRb для дистрибутивного програмування, можна домогтися більш оптимальних результатів.

8.3. Застосування модуля DRb для синхронізації процесів

Модуль DRb є частиною стандартної Ruby бібліотеки і відповідає за можливості дистрибутивного програмування. В основі його ідеї лежить можливість дати доступ до одного Ruby об'єкту будь-якому комп'ютеру в мережі. Результати всіх маніпуляцій з цим об'єктом, його внутрішній стан, видно всім підключеним комп'ютерам, і постійно синхронізовано. В цілому, можливості модуля дуже широкі.

Для вирішення цієї проблеми можна використовувати кортежі Rinda::TupleSpace разом з можливостями DRb, щоб створити модуль Pthread, що відповідає за виконання коду в окремих процесах як на комп'ютері головної програми, так і на інших підключених машинах. Rinda::TupleSpace пропонує доступ до кортежів по імені і, як і об'єкти класу Queue, дозволяють записувати і зчитувати кортежі тільки одному потоку або процесу в один час. Таким чином, дане рішення дозволяє в Ruby MRI виконувати код на кількох ядрах:

```
Pthread::Pthread.new queue: 'fact', code: %{
    1.upto(n).inject(1) { |i, n| i * n }
}, context: { n: 1000 }
```

Код, який необхідно виконати, подається як рядок, тому що у випадку з процедурами DRb передає іншому процесу тільки посилання на неї, і для її виконання користується ресурсами процесу, який цю процедуру створив

8.4. Паралелізм в Ruby on Rails

Сервера для Ruby on Rails і бібліотеки для виконання фонових завдань можна розбити на дві групи. Перша для обробки запитів користувачів або виконання фонових завдань використовує Форки – додаткові процеси. Таким чином, використовуючи MRI і ці сервера і бібліотеки, можливо обробляти паралельно відразу кілька запитів, і виконувати кілька завдань одночасно.

Однак, у цього способу є недолік. Форки процесів копіюють пам'ять процесу, який їх створив, і таким чином сервер Unicorn з трьома «працівниками» може забирати 1ГБ пам'яті, ледь почавши роботу. Теж саме стосується бібліотек для виконання фонових завдань, наприклад Resque.

Творці сервера Puma для Ruby on Rails врахували особливості jRuby і Rubinius, і випустили сервер, орієнтований в першу чергу на ці дві імплементації. На відміну від того ж Unicorn, Puma для одночасної обробки запитів використовує потоки, які вимагають набагато менше пам'яті. Таким чином, Puma буде відмінною альтернативою при використанні в зв'язці разом з jRuby або Rubinius. Тому же принципом збільшено втричі бібліотека Sidekiq.

8.5. Завдання до лабораторної роботи

Завдання для програмування береться з файлу додатку до Лабораторних робіт. Номер варіанта відповідає номеру комп'ютера, на якому працює студент. Звіт до даної лабораторної роботи повинен містити

1. Назву роботи.
2. Тему, мету і завдання до роботи.
3. Вихідний код збережених програм.
4. Знімки екрану (скріншоти) процесу розробки
5. Знімок результатів виконання роботи.
6. Висновки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1 Fayez Gebali. Algorithms and parallel computing. – NY.: John Wiley & Sons, 2011. – 365 p.

2 Bates, Mark. Distributed programming with Ruby. – NY.: Addison-Wesley, 2010. – 260 p.

3 Masatoshi Seki. The dRuby Book. Distributed and Parallel Computing with Ruby. – Texas: Pragmatic Programmers, LLC, 2012. – 266 p.

4 Parallel and Distributed Computing Handbook / Albert Y. Zomaya, Editor. – McGraw-Hill, 2018. – 1179 p.

5 Handbook of Parallel Computing. Models, Algorithms and Applications / Edited By Sanguthevar Rajasekaran, John Reif, 1st Edition. – New York: Chapman and Hall/CRC, 2007. – 1224 p. DOI: https://doi.org/10.1201/9781420011296

6 Жуков І.А., Корочкін О.В. Паралельні та розподілені обчислення. Навч. посіб. –К.: «Корнійчук» , 2005. – 226 с.

7 Ясько, М.М. Навчальний посібник до вивчення курсів “Паралельна обробка даних” та “Мови обчислень та кластерні системи” [Текст] /М.М.Ясько. – Д.: РВВДНУ, 2010. – 76с.

8 Є.Ваврук, О.Лашко Організація паралельних обчислень// Навчальний посібник з дисципліни «Паралельні та розподілені обчислення» для студентів базового напряму 6.0915 — «Комп’ютерна інженерія» — Л.: Вид-во Нац. ун-ту «Львівська політехніка», 2007. — 70 с.

9 Tomas Dave with Chad Fowler and Andy Hunt. Programming Ruby. The Pragmatic Programmers’ Guide: Second Edition. Dallas, Texas. – 883 p.

10 Masatoshi Seki. The dRuby Book. Distributed and Parallel computing with Ruby. English translation by Ptagmatic Programmers LLC, 2012. – 256 p.

11 Flanagan D., Matsumoto Y. The Ruby Programming Language. O'Reilly, 2008. – 429 p..

12 Flanagan, David. The Ruby Programming Language: Everything You Need to Know. 1st Edition. O'Reilly Media, 2008. – 448 p.

13 Hal Edwin Fulton. The Ruby Way. 2nd Edition. Addison-Wesley Professional. 2006. – 839 p.

14 Fitzgerald , Michael James. Learning Ruby: The Language that Powers Rails. 1st Edition. O'Reilly Media. 2007. – 275 p.

15 McGavren, Jay. Head First Ruby: A Brain-Friendly Guide 1st Edition. O'Reilly Media. 2015. – 572 p.

16 Tanenbaum, Andrew S., Maarten van Steen. Distributed Systems. Principles and Paradigms. Second Edition. Pearson Education. Inc. 2007. – 686 p.

17 Black, David A . The Well -Grounded Rubyist . Second Edition . Manning Publications. 2014. – 536 p.

18 Cooper, Peter. Beginning Ruby: From Novice to Professional (Expert's Voice in Open Source) 2nd ed. Edition. Apress. 2009. – 652 p.

19 Dwight, Robert. Ruby: Learn Ruby in 24 Hours or Less - A Beginner’s Guide To Learning Ruby Programming Now (Ruby, Ruby Programming, Ruby Course). CreateSpace Independent Publishing Platform. 2016. – 110 p.

20 [Електронний ресурс]. Режим доступу: URL: https://www.codica.com/blog/top-ruby-gems-we-cant-live-without/
21 [Електронний ресурс]. Режим доступу: URL: https://parallel.ru/tech/tech_dev/ifaces.html#pvm
22 [Електронний ресурс]. Режим доступу: URL: https://rubygarage.org/blog/the-best-technology-for-MVP

Навчальне видання

Коротенко Григорій Михайлович
Коротенко Леонід Михайлович

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
до виконання лабораторних робіт з дисципліни
«Технології розподілених систем та паралельних обчислень»
на мові Ruby
для студентів спеціальності
126 «Інформаційні системи та технології»

Видано в світ
у Національному технічному університеті
«Дніпровська політехніка».
Свідоцтво про внесення до Державного реєстру ДК № 1842
4960050, м. Дніпро, просп. Д. Яворницького, 19