**Комментарии и демонстрация результата**

*Комментарием* называется часть программного кода, пропускаемая при обработке (интерпретации или компиляции).

В Ruby знаком начала комментария служит [#](http://ru.wikipedia.org/wiki/%D0%9E%D0%BA%D1%82%D0%BE%D1%82%D0%BE%D1%80%D0%BF). Всё, что между ним и концом строки пропускается. Пример:

puts 2 + 2 # это комментарий

puts "Привет!" # тоже комментарий

Многострочные комментарии помещаются между словами =begin и =end

=begin

Это длинный комментарий

Очень длинный

=end

puts "Привет!"

Результат иллюстрируемого кода будет располагаться после последовательности #=>. Пример:

puts 2 + 2 #=> 4

puts "Привет" #=> Привет

**Вывод на экран**

В Ruby есть много методов вывода: print, printf, p, puts, .display и другие. Но мы использовать будем два:

метод puts. После вывода строки осуществляет переход на новую. Если приходится выводить объект, не являющийся строкой, то puts вызывает метод .to\_s для преобразования его к строке;

метод p. Имеет самое короткое название, из-за чего часто используется для отладки. Так же, как и puts, делает перевод на новую строку после вывода. Перед выводом любого объекта (в том числе и строки) на экран, вызывает метод .inspect.

Примеры вывода на экран:

puts [1, 2, 3, 4] #=> 1\n2\n3\n4

# \n означает перевод строки

p [1, 2, 3, 4] #=> [1, 2, 3, 4]

puts "Hello!" #=> Hello!

p "Hello!" #=> "Hello!"

puts 5 #=> 5

p 5 #=> 5

Как видно из примера, результаты во время вывода строк и массивов существенно различаются. Если вас не смущают кавычки в результате вывода, то смело используйте p, если смущают, то puts.

**Конструкция Если**if условия  
  выражения  
end  
  
**Конструкция противоположная Если**unless условия  
  выражения  
end  
  
  
Также в обеих конструкциях можно использовать else if в руби пишется elsif, можно использовать также обычное else. Пример полного блока

if условие 1  
  выражение  
elsif условие 3  
  выражение  
else # если не выполнилось условие 1 и условие 2  
  выражение  
end

В одном блоке if если выполнилось условие, то следующие условия не проверяются.  
  
Цикл while пока выполняется пока выполняется условие заданное после while

a = 1  
while a<10  
 puts a  
end

***Цикл  Loop***

Простейшая конструкция цикла в Ruby это метод loop.

Ruby вместе с разнообразием циклов и прочих управляющих структур поставляет нам управляющие работой цикла ключевые слова, что позволяет нам создать цикл более полезный нежели тот, который я приводил выше. Эти слова – это: break, next, redo. Ключевое слово break позволяет нам выйти из цикла в любой момент.

loop do

i+=1

print "#{i}"

break if i==10

end

Данный код напечатает: 1 2 3 4 5 6 7 8 9 10

В данном примере цикл прекращает работу после того, как i станет равной 10.

[*Переменные*](http://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) используются, чтобы сохранить промежуточный результат вычислений. Имя переменной в Ruby должно:

* начинаться с буквы или знака подчёркивания;
* состоять из латинских букв, цифр и знака подчёркивания.

Примеры переменных:

array

variable

another\_variable

3element # неправильное имя. Начинается с цифры

\_3element # а вот как можно

\_\_ # немного странное, но допустимое имя переменной

Переменная может иметь имя не только латинское, но и русское. Для этого, правда, требуется, чтобы весь текст программы был написан в кодировке UTF-8, а интерпретатор запускался с параметром -KU.

[Присваивание](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D1%81%D0%B2%D0%B0%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) осуществляется знаком равенства (=), вот так:

array = [1, 2, 3, 4]

variable = array + [1, 2]

another\_variable = variable - array

То, что справа от =, прежде чем стать значением переменной, обычно полностью вычисляется. Наша переменная variable будет содержать массив [1, 2, 3, 4, 1, 2], потому что (ибо - это слово в значении "для того что бы") именно таков результат действия array + [1, 2]. Плюс (+) с массивами поступает именно так: прицепляет второй массив в хвост первого. Переменная another\_variable теперь содержит пустой массив, поскольку операция минус убирает все вхождения элементов массива array из массива variable.

Переменные *указывают* на объект

Фокус-покус:

girlfriend = "Даша"

goes\_on\_a\_visit = girlfriend

puts girlfriend #=> Даша, разумеется

goes\_on\_a\_visit[0] = "М" # меняем первую (номер ноль) букву у переменной-строки

puts girlfriend #=> Маша

# На первый взгляд странно и неожиданно

Значение, возвращаемое первой переменной, изменилось потому, что в Ruby переменные содержат лишь *ссылку* на то, что вы им присваиваете.

Сами данные (*объект*) лежат где-то в другом месте. Ввиду этого естественно, что при прямом изменении самого объекта, на который указывает переменная, все другие переменные, указывающие на этот объект, будут возвращать изменённое значение (также будут изменяться).

Чтобы girlfriend наша осталась "Даша", надо в переменную занести её [клон](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%BE%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29):

girlfriend = "Даша"

goes\_on\_a\_visit = girlfriend.clone

goes\_on\_a\_visit[0] = "М" # Но изменили мы лишь клон. Дома в сохранности сидит настоящая:

puts girlfriend #=> Даша

Можно создавать копии объектов ещё методом .dup. Разница между ними будет понятна позже.

Для безвредного присваивания новых значений переменным их редко приходится клонировать, ибо большинство методов делают это и так. Даже если вы просто присвоите переменной новое значение, Ruby создаст объект с новым значением и поместит в (уже существующую) переменную ссылку на тот объект:

# …

goes\_on\_a\_visit = "Аристарх" # Создаётся новый объект, переменная переводится на него

p girlfriend #=> "Даша"

**%Q**

Альтернатива строке в двойных кавычках. Полезно, когда в строке имеются собственные кавычки — вместо экранирования обратным слешем можно использовать %Q:

%Q(Mark said: "Tom said: "#{what\_tom\_said}"")

=> "Mark said: "Tom said: "Hello!"""

Скобочки здесь “(…)” могут быть заменены на любой не алфавитный символ:

>> %Q!Mark said: "Tom said: "#{what\_tom\_said}""!

>> %Q[Mark said: "Tom said: "#{what\_tom\_said}""]

>> %Q+Mark said: "Tom said: "#{what\_tom\_said}""+

Так же можно использовать:

>> %/Mark said: "Tom said: "#{what\_tom\_said}""/

=> "Mark said: "Tom said: "Hello!"""

**%q**

Используется для строки в одинарных кавычках. Синтаксис похож на %Q, но уже без подстановки выражений и экранирования:

>> %q(Mark said: 'Tom said: '#{what\_tom\_said} ' ')

=> "Mark said: 'Tom said: '\#{what\_tom\_said} ' '"

**%W**

Используется для элементов массива в двойных кавычках. Синтаксис схож с %Q:

>> %W(#{foo} Bar Bar\ with\ space)

=> ["Foo", "Bar", "Bar with space"]

**%w**

Используется для элементов массива в одинарных кавычках.

>> %w(#{foo} Bar Bar\ with\ space)

=> ["\#{foo}", "Bar", "Bar with space"]