Санкт-Петербургский Государственный Электротехнический Университет «ЛЭТИ» имени В.И. Ульянова (Ленина)

Кафедра Автоматики и Процессов Управления

**РЕФЕРАТ**

НА ТЕМУ

«Ruby-язык программирования»

**Выполнил:** студент группы 3371

Потапова В.Э.

**Преподаватель:** Кондратьева Н.Ю.

Санкт-Петербург

2015

**Содержание:**

1. **Общие сведен……………………………………………………………………………………………….3**
2. **История создания и развития……….……………………………………………………………..3**
3. **Хронология выхода версий………………………………………………………………………….3**
4. **Семантика…………………………………………………………………………………………………….6**
5. **Возможности Ruby……………………………………………………………………………………….6**
6. **Сравнение Ruby с другими языками…………………………………………………………..7**
7. **Основы………………………………………………………………………………………………………….8**

**7.1 Вызов метода………………………………………………………………………………………….10**

**7.2 Прокси классы…………………………………………………………………………………………10**

**7.3 Синглтон классы………………………………………………………………………………………11**

**7.4 Self и текущий класс………………………………………………………………………………..13**

**7.5 Верхний уровень……………………………………………………………………………………..14**

**7.6 Определение класса………………………………………………………………………………..14**

**7.7 Определение метода……………………………………………………………………………….14**

**7.8 Цепочки наследования…………………………………………………………………………….16**

**7.9 Вывод………………………………………………………………………………………………………..17**

**8. Заключение……………………………………………………………………………………………………..18**

**Общие сведения**

**Ruby** - динамический, рефлективный, интерпретируемый высокоуровневый язык программирования для быстрого и удобного объектно-ориентированного программирования. Язык обладает независимой от операционной системы реализацией многопоточности, строгой динамической типизацией, сборщиком мусора и многими другими возможностями. По особенностям синтаксиса он близок к языкам Perl и Eiffel, по объектно-ориентированному подходу — к Smalltalk. Также некоторые черты языка взяты из Python, Lisp, Dylan и Клу.

**История создания и развития**

Создатель Ruby — Юкихиро Мацумото (Matz) — интересовался языками программирования, ещё будучи студентом, но идея о разработке нового языка появилась позже. Ruby начал разрабатываться 23 февраля 1993 года и вышел в свет в 1995 году.

Название навеяно языком Perl, многие особенности синтаксиса и семантики из которого заимствованы в Ruby: англ. *pearl* — «жемчужина», *ruby* — «рубин».

Целью разработки было создание «настоящего объектно-ориентированного», лёгкого в разработке, интерпретируемого языка программирования.

В Японии Ruby стал популярным с момента появления первой общедоступной версии в 1995 году, однако наличие документации только на японском языке сдерживало его дальнейшее распространение. Лишь в 1997 году появилось описание Ruby на английском языке, а в 1998 году открылся форум «ruby-talk». Это положило начало росту известности языка в остальном мире. Издано несколько книг на различных языках, в том числе на русском. Сейчас Ruby входит в большинство дистрибутивов ОС Linux, поставляется вместе с Mac OS X, доступен пользователям других операционных систем.

**Хронология выхода версий**

|  |  |  |
| --- | --- | --- |
| **Название версии** | **Дата выхода** | **Примечания** |
| 0.06 | 7 января 1994 | Первая версия, указанная в Changelog’ах |
| 1.0-961225 | 25 декабря 1996 | Данная версия следовала сразу за версией 0.99.4-961224, выпущенной накануне. Номер после числа 1.0 — дата выпуска версии. Новые версии линейки 1.0 выходили ещё год (до 1.0-971225). |
| 1.1 alpha0 | 13 августа 1997 | Альфа-версии выходили вплоть до 7 октября 1997 (1.1 alpha9) |
| 1.1b0 | 4 декабря 1997 | Следующая версия после 1.1 alpha9. 27 февраля 1998 вышла версия 1.1b9, затем вплоть до середины 1998 выходили экспериментальные выпуски с обозначением вида 1.1b9\_31 (версия 1.1b9\_31 была выпущена, но в документации не отмечена). |
| 1.1c0 | 17 июля 1998 | Данная версия следовала за версией 1.1b9\_31. Модификации этой версии выходили вплоть до 26 ноября 1998 (1.1c9). |
| 1.1d0 (pre1.2) | 16 декабря 1998 | Данная версия следовала за версией 1.1c9. 22 декабря 1998 была выпущена экспериментальная версия 1.1d1, завершившая данную линейку. |
| 1.2 (stable) | 25 декабря 1998 | В дальнейшем выходили модификации данной версии вплоть до версии 1.2.5, выпущенной 13 апреля 1999 года. 21 июня 1999 года была выпущена версия 1.2.6, объявленная как финальная версия 1.2 (1.2 final). 15 июля 1999 года вышла переупакованная (repacked) версия 1.2.6. |
| 1.3 (development) | 24 декабря 1998 | Отдельная ветка модификаций, разрабатываемая независимо от линейки 1.2 (по аналогии с ядром ОС Linux). Первая версия была объявлена как версия для разработки (development version) и следовала за версией 1.1d1. В дальнейшем последовало множество промежуточных модификаций: ruby-1.3.1-990215 — ruby-1.3.4-990625, после чего от указания даты в номере отказались и выпустили 1.3.5 — 1.4 alpha (15 июля 1999), 1.3.6 — 1.4 alpha (28 июля 1999), 1.3.7 — 1.4 beta (6 августа 1999). |
| 1.4.0 (stable) | 13 августа 1999 | Данная версия появилась через несколько дней после выхода 1.3.7 — 1.4 beta. В дальнейшем выходили новые модификации вплоть до версии 1.4.6, вышедшей 16 августа 2000 года. |
| 1.5.0 (development) | 20 ноября 1999 | Данная линейка предназначалась исключительно для проверки различных нововведений при разработке. Модификации данной линейки доступны исключительно в репозитории проекта и, соответственно, сборки данной версии на официальный сервер не выкладывались. |
| 1.6.0 (stable) | 19 сентября 2000 | В дальнейшем выпускались модификации этой версии вплоть до версии 1.6.8 (24 декабря 2002). 21 сентября 2005 года был выпущен патч для версии 1.6.8. |
| 1.7.0 (development) | 24 февраля 2001 | Данная линейка предназначалась исключительно для проверки различных нововведений при разработке. Модификации данной линейки доступны исключительно в репозитории проекта и, соответственно, сборки данной версии на официальный сервер не выкладывались. |
| 1.8.0 (stable) | 4 августа 2003 | В дальнейшем последовало большое число модификаций, которые выходят до сих пор (1 января 2011 года), например, промежуточная версия 1.8.7-p330 вышла 24 декабря 2010 года. |
| 1.9.0 (development) | 25 декабря 2007 | Изначально экспериментальная ветка, созданная для практической проверки ряда нововведений. |
| 1.9.3 (stable) | 31 октября 2011 | Отличия от 1.9.2 — существенны. |
| 2.0.0 (stable) | 24 февраля 2013 |  |
| 2.1.0 (stable) | 25 декабря 2013 |  |
| 2.2.0 (stable) | 25 декабря 2014 | Поддержка Unicode 7.0, добавлена сборка мусора для объектов типа Symbol. |

**Семантика**

Ruby — полностью объектно-ориентированный язык. В нём все данные являются объектами, в отличие от многих других языков, где существуют примитивные типы. Каждая функция — метод.

Ruby использует вызов по соиспользованию (*call-by-sharing*), хотя в сообществе Ruby часто говорят, что он использует вызов по ссылке. Для программиста, привыкшего к распространённым гибридным языкам программирования, некоторые эффекты такого решения могут показаться неожиданными. Например:

a = "abcdefg"

b = a

b *#=> "abcdefg"*

a[4] = 'R'

b *#=> "abcRefg"*

то есть при изменении значения переменной *a*, неявно изменилось и значение *b*, так как они содержат ссылку на один объект. То есть механизм присваивания действует одинаково для всех объектов, в отличие от языков типа C, Object Pascal, где присваивание может означать как копирование значения, так и копирование ссылки на значение.

Ruby не поддерживает множественное наследование, но вместо него есть мощный механизм примесей. Все классы (напрямую или через другие классы) выведены из класса Object, следовательно, любой объект может использовать определённые в нём методы (например, class, to\_s, nil?). Процедурный стиль также поддерживается, но все глобальные процедуры неявно являются закрытыми методами класса Object.

Ruby является мультипарадигменным языком: он поддерживает процедурный стиль (определение функций и переменных вне классов), объектно-ориентированный (всё — объект), функциональный (анонимные функции, замыкания, возврат значения всеми инструкциями, возврат функцией последнего вычисленного значения). Он поддерживает отражение, метапрограммирование, информацию о типах переменных на стадии выполнения.

**Возможности Ruby**

* Имеет лаконичный и простой синтаксис, частично разработанный под влиянием Ада, Eiffel и Python.
* Позволяет обрабатывать исключения в стиле Java и Python.
* Позволяет переопределять операторы, которые на самом деле являются методами.
* Полностью объектно-ориентированный язык программирования. Все данные в Ruby являются объектами в понимании Smalltalk. Единственное исключение — управляющие конструкции, которые в Ruby, в отличие от Smalltalk, не являются объектами. Например, число «1» — это экземпляр класса Fixnum. Также поддерживается добавление методов в класс и даже в конкретный экземпляр во время выполнения программы.
* Не поддерживает множественное наследование, но вместо него может использоваться концепция «примесей», основанная в данном языке на механизме модулей.
* Содержит автоматический сборщик мусора. Он работает для всех объектов Ruby, в том числе для внешних библиотек.
* Создавать расширения для Ruby на Си очень просто частично из-за сборщика мусора, частично из-за несложного и удобного API.
* Поддерживает замыкания с полной привязкой к переменным.
* Поддерживает блоки кода (код заключается в { … } или do … end). Блоки могут использоваться в методах или преобразовываться в замыкания.
* Целые переменные в Ruby автоматически конвертируются между типами Fixnum (32-разрядные) и Bignum (больше 32 разрядов) в зависимости от их значения, что позволяет производить целочисленные математические расчёты со сколь угодно большой точностью.
* Не требует предварительного объявления переменных, но для интерпретатора желательно, чтобы переменным присваивалось пустое значение nil (тогда интерпретатор знает, что идентификатор обозначает переменную, а не имя метода). Язык использует простые соглашения для обозначения области видимости. Пример: просто var — локальная переменная, @var — переменная экземпляра (член или поле объекта класса), @@var — переменная класса, $var — глобальная переменная.
* В Ruby непосредственно в языке реализованы многие шаблоны проектирования, так, например, «одиночка» (singleton) может быть (хотя и не обязан) реализован добавлением необходимых методов к одному конкретному объекту .
* Может динамически загружать расширения, если это позволяет операционная система.
* Имеет независимую от ОС поддержку невытесняющей многопоточности.
* Перенесён на множество платформ. Он разрабатывался на Linux, но работает на многих версиях Unix, DOS, Microsoft Windows (в частности, Win32), Mac OS, BeOS,OS/2 и т. д.

**Сравнение Ruby с другими языками**

Начнем с языка Java:

\* Ruby -- интерпретируемый язык;

\* в Ruby все является объектом (в Java есть типы int и Integer, что создает определенные неудобства);

\* переменные в Ruby не являются статически типизированными и не требуют объявления;

\* модули (modules) в Ruby позволяют с помощью <<миксинов>> (mixins) конструировать подобие интерфейсов (interfaces) языка Java, допуская при этом в них реализацию методов.

Теперь сравним с языком Perl:

\* Ruby значительно легче освоить, на нем легче программировать, а написанные программы проще сопровождать;

\* в Ruby префиксы (@, $, @@) описывают область видимости (scope), а не тип переменной;

\* Ruby позаимствовал из языка Perl регулярные выражения, переменную $\_ и многое другое.

Завершим сопоставление Ruby с другими языками рассмотрением одного из ближайших <<конкурентов>> - языка Python. Это сравнение особенно интересно потому, что именно Python рассматривается сейчас многими, как хороший кандидат на роль первого языка программирования. Итак, сравним Ruby с языком Python:

\* управляющие конструкции и методы в языке Ruby завершаются ключевым словом end, в то время как Python использует так называемый <<двумерный>> синтаксис, когда признаком завершения является изменение количества лидирующих пробелов в очередной строке программы;

\* вместо self в Ruby для обозначения переменных экземпляра используется префикс @;

\* в Ruby, в отличие от языка Python, понятия типа и класса являются синонимами;

\* Python не поддерживает наследования и не позволяет добавлять методы к существующим типам;

\* используемый в Ruby алгоритм сборки мусора позволяет проще писать реализации методов на языке C;

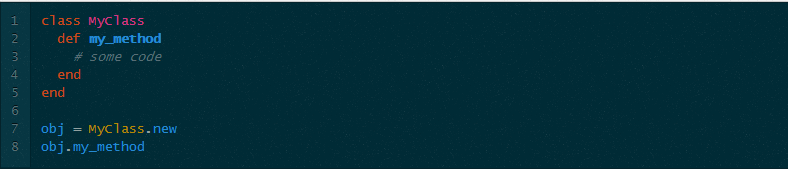
\* расширения для Ruby, написанные на C/C++ позволяют определять новые классы;

\* зачастую Ruby быстрее, чем Python.

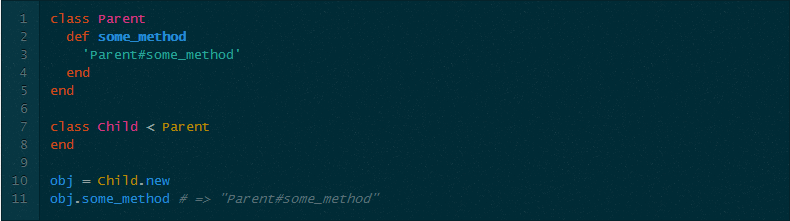
7.Объектная модель Ruby

**Основы**

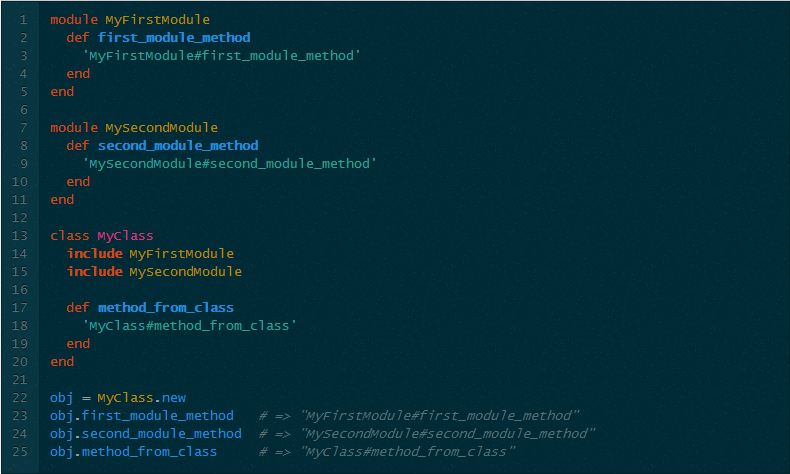
Ruby является полностью объектно-ориентированным языком: числа, строки, регулярные выражения, массивы - это все объекты определенных классов. Класс определяет поведение объекта - он содержит все методы, доступные его объектам (инстансам).



Традиционно в ООП используется наследование классов - методы класса-родителя доступны объектам дочерних классов (а также их дочерних классов и т.д.). Разумеется, в ruby этот механизм тоже не забыт:



Однако, множественное наследование не поддерживается: у каждого класса может быть только один родительский класс - за исключением BasicObject (в 1.8 и ранее Object), который является корнем иерархии и не имеет родителя. В случаях, когда нужно предоставить какое-то кол-во методов нескольким классам, но нежелательно пользоваться наследованием (скажем, когда нужно наследоваться от какого-то другого класса), удобно использовать механизм примесей (mixins). Это происходит следующим образом: в модуле определяются методы, потом определенный класс подключает этот модуль - и объекты класса получают все методы, определенные в модуле. При этом кол-во включаемых в класс модулей ограничено только здравым смыслом.

****

**Вызов метода**

Итак, каждый объект является инстансом определенного класса, а класс в свою очередь имеет цепочку наследования, которая восходит вверх до BasicObject.

Объект содержит какое-то количество инстанс-переменных (например, объект класса Person может иметь инстанс-переменную @name) и ссылку на свой класс.

Класс содержит определенное количество инстанс-методов (так я буду называть методы, которые класс предоставляет своим объектам и объектам дочерних классов, чтобы не путать их с классовыми методами - такими как Time.now), некоторое число подключенных модулей и ссылку на родительский класс.

Посмотрим, что происходит, когда вызывается определенный метод:

1. определяется объект-получатель вызова
2. в иерархии классов ищется сам метод

Начнем с получателя. В любом участке кода всегда есть текущий объект - его возвращает псевдопеременная self. У текущего объекта есть два основных значения: при обращении к инстанс-переменным ruby будет искать их в текущем объекте, и при вызове метода без указания получателя он будет вызван на текущем объекте. При вызове метода с указанием получателя, соответственно, получателем будет указанный объект.

Так вот. Как только выполнение попадает внутрь вызванного метода, self становится этим объектом-получателем и остается им до тех пор, пока не будет вызван метод с другим объектом-получателем, либо пока метод не будет полностью выполнен (а также в некоторых других случаях, которые будут описаны далее). Все это время обращения к инстанс-переменным будут работать с текущим объектом, и вызовы методов без указания получателя будут попадать также в текущий объект.

Теперь посмотрим на поиск метода в иерархии классов. У нас уже есть объект-получатель вызова, а он содержит ссылку на свой класс. Если класс содержит метод с таким названием, то этот метод используется; иначе поиск продолжается в классе-родителе, потом в его родителе и т.д. - вплоть до BasicObject.

Если же метод так и не был найден, будет вызван метод method\_missing с параметрами, первым из которых будет название вызванного метода, далее аргументы, переданные при вызове, и блок при его наличии. Вызов происходит по той же схеме - сначала method\_missing ищется в классе объекта, потом в его родителе и т.д. Стандартный BasicObject#method\_missing выбрасывает NoMethodError, если вызов добирается до него.

**Прокси-классы**

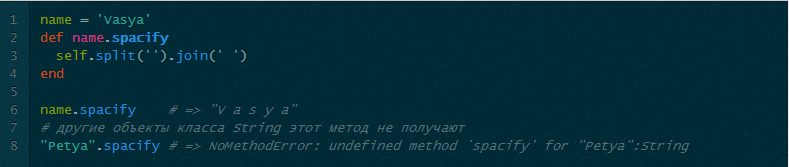
Но как в эту схему вписываются модули, подключенные в классы из цепочки наследования? Их методы также должны быть видимы объекту класса, который стоит ниже в цепи (если считать, что BasicObject стоит на самом верху).

Дело в том, что при включении класса руби создает анонимный класс, и помещает его в цепочку наследования прямо над включившим этот модуль классом. Такие анонимные классы часто называют прокси-классами. Соответственно, при поиске метода, определенного в модуле, он будет найден в анонимном прокси-классе, и все произойдет так, как если бы метод был определен в одном из настоящих классов.

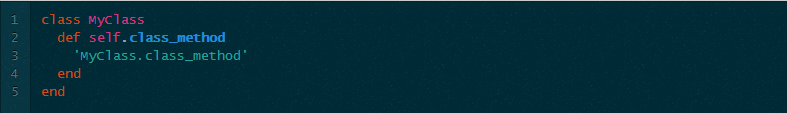


**Синглтон-классы**

В ruby есть специальная форма определения метода: def object.method\_name. Созданный таким образом метод называется синглтон-методом - он определен только для этого конкретного объекта.

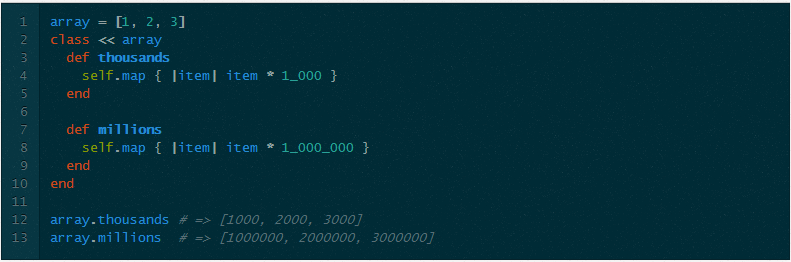


Именно благодаря этой особенности в ruby работают классовые методы: каждый класс является объектом класса Class, который предоставляет лишь необходимый минимум методов - new, superclass и т.д. Если же нам нужно определить классовый метод (который будет инстанс-методом с точки зрения класса Class), обычно мы задаем его следующим образом:



Разумеется, в данном случае метод class\_method становится доступен только MyClass, и никакому другому классу (на самом деле он также будет доступен дочерним классам MyClass, но об этом чуть позже).

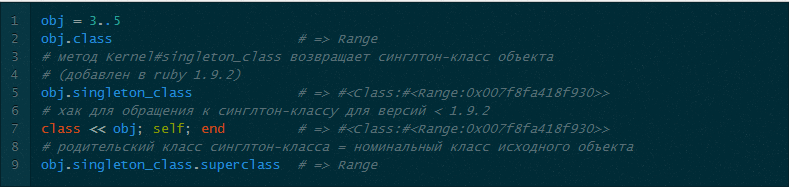
Часто для определения сразу нескольких классовых методов используется конструкция class << self внутри определения класса; в этом случае методы, определенные в обычной форме (def method\_name), становятся классовыми методами. То же самое применимо и к обычным объектам - в общем случае конструкция выглядит как class <<object:



Это все может выглядеть как магия, но на самом деле ruby руководствуется единой логикой. Посмотрим внимательнее, что тут происходит.

Как уже говорилось выше, объект - это просто группа инстанс-переменных и ссылка на класс, который в свою очередь содержит некоторое кол-во инстанс-методов. Но синглтон-методы видны только одному конкретному объекту, поэтому не могут находиться в классе объекта; в то же время метод не может находиться в объекте, так как только классы и модули могут содержать методы (предоставляя их своим объектам и объектам дочерних классов).

Решением этой загадки является синглтон-класс (также встречается термин eigenclass). Это прослойка между объектом и его номинальным классом (тем самым, который можно получить через object.class). Все синглтон-методы объекта попадают в его синглтон-класс, и уже он является “реальным” классом объекта. В то же время, чтобы не терять возможность вызывать инстанс-методы, определенные в номинальном классе, синглтон-класс наследуется от него, поэтому поиск метода при его вызове начинается с синглтон-класса, дальше поднимается в номинальный класс, в его родителя и дальше вверх до BasicObject.



Наследование синглтон-классов классов устроено несколько сложнее. Классы наследуют у своих родителей не только инстанс-методы, но и классовые методы - но, как мы знаем, они содержатся в синглтон-классе класса, который при вышеописанной схеме наследования не попадает в цепочку. Поэтому в ruby синглтон-класс класса наследуется от синглтон-класса его родителя. Звучит немного запутанно, поэтому лучше посмотреть на примере:



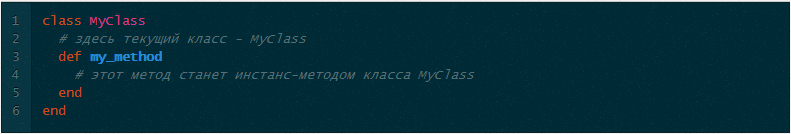
К сожалению (а, может, и к счастью), синглтон-методы скрыты от глаз программиста в методах, инспектирующих цепочку наследования - ни Kernel#class, ни Module#ancestorsих не показывают. Но, как видно из последнего примера, с помощью Object#singleton\_class и Class#superclass это можно обойти.

**Self и текущий класс**

В каждой точке кода определены так называемые “текущий объект” и “текущий класс”.

Про текущий объект мы уже поговорили - это объект, который доступен через псевдопеременную self; к нему адресуются все вызовы методов без указания получателя и в нем ищутся все запрошенные инстанс-переменные.

Текущий класс - это тот класс, инстанс-методом которого становится определенный в этом месте метод.

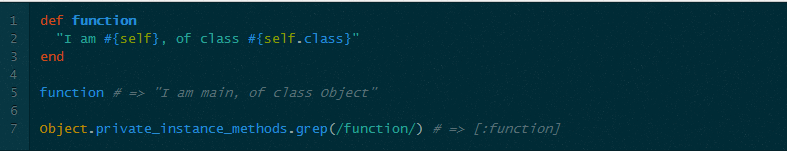


К сожалению, в отличие от текущего объекта, который можно узнать через self, текущий класс можно отслеживать только по коду.

Посмотрим все типичные случаи, имеющие собственные правила определения self и текущего класса.

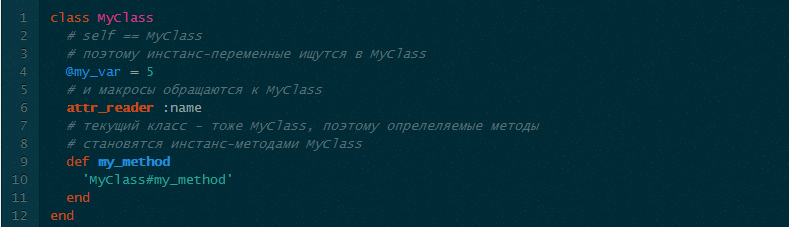
**Верхний уровень**

До того, как мы входим в определение какого-либо класса, self указывает на main, а текущим классом является Object. Это объясняет тот факт, что методы, определяемые на верхнем уровне, вызываются из любого участка кода, причем без получателя (такой метод будет приватным, т.е. его нельзя вызывать с явным получателем; а self указывает на объект класса Object либо его потомка, поэтому метод будет доступен).



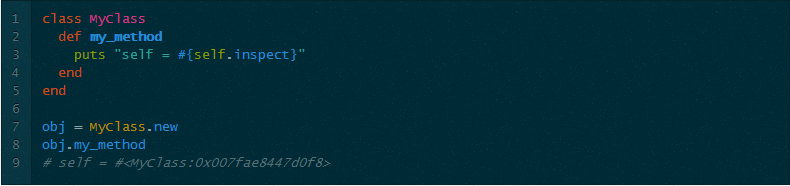
**Определение класса**

В начале объявления класса self и текущий класс меняются на объявляемый класс. Пожалуй, это самый очевидный пример использования текущего класса - определяемые методы становятся инстанс-методами класса, в определении которого это происходит. Классовые макросы, вроде attr\_accessor, вызываются без указания получателя и попадают в self, т.е. применяются на объявляемый класс.



**Определение метода**

В объявлении метода self указывает на объект, являющийся получателем при вызове этого метода. Это, в свою очередь, самый очевидный пример смены текущего объекта.



Текущий класс в определении метода не меняется; внутри определения метода можно определить второй метод, и при вызове первого второй станет методом того же класса:

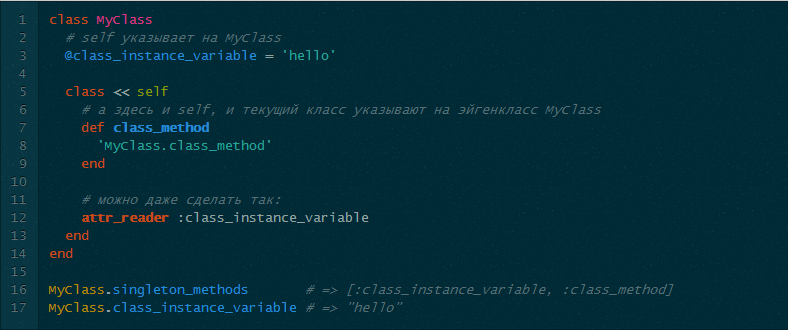


**class << object**

Как уже говорилось чуть выше, у каждого объекта есть свой синглтон-класс - собственный класс объекта, содержащий все его синглтон-методы и наследующий от его номинального класса. Для определения синглтон-метода достаточно формы def self.method\_name, но если необходимо поработать с синглтон-классом более плотно, на помощь приходит конструкция class << object. Как только мы попадаем внутрь нее, self и текущий класс меняются на синглтон-класс нашего объекта.

Когда речь идет об обычном объекте, мы можем определить для него сразу несколько синглтон-методов:

В более распространенной ситуации, когда class << self используется в определении класса, в роли объекта выступает сам класс, и мы попадаем в синглтон-класс класса:



**BasicObject#instance\_eval / Module#class\_eval**

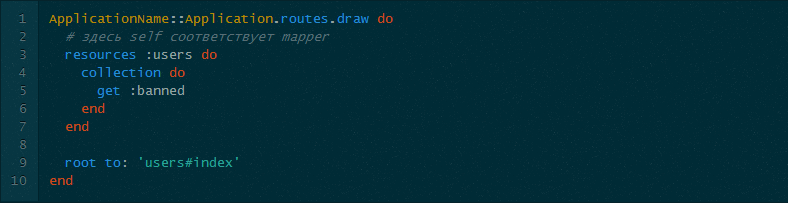
Часто возникает необходимость открыть существующий объект и изменить его “внутренности” - изменить инстанс-переменные, не имеющие attr\_writer и недоступные снаружи; добавить пару синглтон-методов разом; а если объект является классом - то и инстанс-методы (синтаксис class MyClass ... end не подходит, если название класса определяется в рантайме, и на класс просто указывает переменная). Для таких случаев есть BasicObject#instance\_eval и Module#class\_eval. Они также меняют self и текущий класс.

**BasicObject#instance\_eval**

instance\_eval - это интерпретация кода в контексте определенного объекта. Внутри блока self соответствует самому объекту, а текущий класс - синглтон-классу объекта:

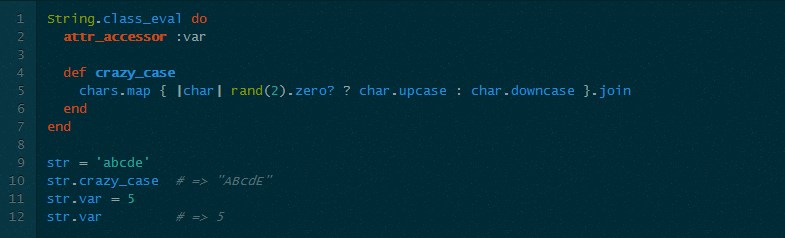
Впрочем, синглтон-методы обычно определяют в конструкции class << object.

Интересное достоинство instance\_eval - в лаконичных DSL, где self, указывающий на нужный разработчику объект, принимает на себя все вызовы методов без указания получателя. Например, так устроен механизм задания маршрутов в роутере rails - вызывается метод ApplicationName::Application.routes.draw, который передает блок методу eval\_block, а тот в свою очередь запускает instance\_exec на объекте mapper - и все методы resources, get, post, member, collection вызываются с этим объектом в качестве получателя:



**Module#class\_eval**

Метод class\_eval, как и его синоним module\_eval, определен в классе Module. В блоке, который передается в class\_eval, self и текущий класс указывают на класс/модуль, на котором вызван этот метод. Фактически, это аналог повторного открытия класса, которое дает возможность обыкновенного monkeypatching: мы можем переопределять инстанс-методы класса и работать с инстанс-переменными самого класса.



**Цепочки наследования**

Мы уже знаем, что при вызове метода он ищется по всей цепочке наследования классов, начиная с синглтон-класса объекта-получателя и до BasicObject. Рассмотрим все случаи наследования классов:

* При объявлении нового класса:
  + если указан класс-родитель, то новый класс наследуется от него
  + если родитель не указан, новый класс наследуется от Object
* При включении модуля в класс создается прокси-класс, содержащий инстанс-методы модуля, и помещается в цепочку прямо над включающим классом
* Синглтон-класс обычного объекта (не класса и не модуля) наследуется от номинального класса этого объекта (класс, который возвращает object.class)
* Синглтон-класс класса наследуется от синглтон-класса родителя исходного класса (кроме синглтон-класса BasicObject, который наследуется от Class)

Таким образом, методы любого объекта состоят из его синглтон-методов, инстанс-методов модулей, подключенных в его синглтон-класс, инстанс-методов его номинального класса и подключенных в него модулей, потом родителя этого класса и его подключенных модулей, и так далее до BasicObject.

При подключении в один класс нескольких модулей порядок в цепочке наследования определяется порядком подключения: подключенный последним модуль будет стоять сразу после класса, его подключившего; далее предпоследний подключенный модуль итд; т.е., например, последний модуль может переопределить метод, заданный в предпоследнем.

Классовые методы любого класса, в свою очередь, состоят из синглтон-методов этого класса и инстанс-методов модулей, подключенных в его синглтон-класс, далее синглтон-методов класса-родителя и инстанс-методов его модулей, и т.д. до синглтон-методов BasicObject; потом инстанс-методов классов Class, Module и Object, модуля Kernel и наконец класса BasicObject (так как цепочка наследования синглтон-класса BasicObject выглядит следующим образом: #BasicObject < Class < Module < Object < Kernel < BasicObject).

**Выводы**

Резюмируя все вышеописанное:

* Любой объект содержит группу инстанс-переменных и ссылку на класс.
* Любой класс или модуль содержит группу инстанс-методов, и предоставляет их всем объектам, в чьих цепочках наследования он находится; также любой класс содержит ссылку на родительский класс.
* Любой объект может иметь собственный синглтон-класс, содержащий его синглтон-методы; он наследуется от номинального класса исходного объекта.
* Классы и модули являются объектами, и обладают всеми свойствами объектов (имеют свой класс и полученные от него методы, также могут иметь синглтон-класс).
* Любой класс имеет свою цепочку наследования - сам класс, его родительский класс, родительский класс родительского класса и т.д. до корня классовой иерархии - класса BasicObject.
* При подключении модуля в класс он помещается в цепочку наследования прямо над этим классом.
* Синглтон-класс объекта наследуется от номинального класса объекта; за исключением синглтон-классов классов, каждый из которых наследуется от синглтон-класса родителя исходного класса.
* При вызове метода любого объекта этот метод ищется во всех классах цепочки наследования класса исходного объекта, начиная с синглтон-класса и заканчивая BasicObject.
* При обращении к инстанс-переменной она ищется в текущем объекте (self); при вызове метода без указания получателя он будет вызван на текущем объекте.
* При объявлении метода в обычной форме (def method\_name) он становится инстанс-методом текущего класса;
* При объявлении метода с указанием получателя (def object.method\_name) он становится синглтон-методом указанного объекта.
* Указатели на текущий объект (self) и текущий класс меняются в зависимости от контекста.

**Заключение**

Этот язык, несомненно, является одним из лучших в качестве первого языка программирования. Быстрый цикл разработки (редактирование - запуск - редактирование), использование интерпретатора, изначальная объектно-ориентированность, нетипизированные переменные, которые не требуют объявления, - все это позволяет учащимся сконцентрировать свое внимание на общих принципах программирования.

Не менее важны мультиплатформенность Ruby и его принадлежность к миру свободно распространяемого ПО. Еще один весомый аргумент в его пользу - возможность практического использования языка в самых разных областях, что не позволит впоследствии профессионалу, который вырастет из новичка, пожалеть о напрасно потраченном времени.

Можно заметить, что значительному числу перечисленных выше требований удовлетворяют и Python, и Java, и C++ и даже (в меньшей степени) Delphi.

Попробуем сначала сравнить языки Ruby и Java, посмотрев на них глазами человека, только начинающего знакомиться с миром программирования. Для такого человека чрезвычайно важна ясность и краткость программ, отсутствие в них странных и непонятных <<магических>> слов. А именно из них и состоит в основном Java-вариант традиционной первой программы!

public class Hello {

public static void main(String[] args) {

System.out.println("Здравствуй, мир!");

}

}

Аналогичную программу на Ruby долго объяснять не придется:

puts "Здравствуй, мир!"

Ruby в этом смысле очень дружелюбный язык. Программа, написанная новичком, может быть очень короткой и не содержать ничего лишнего, а для создания такой программы необходим, в основном, только здравый смысл.

В отличие от языка Java в Ruby нет проблемы <<больших>> чисел. Приведенные выше программы вычисления факториала числа или чисел Фибоначчи, будучи написанными на Ruby, всегда дают верный результат. В случае же языка Java ограничения на диапазон представимых целых чисел приводят к нелепым (с точки зрения ученика школы или студента первого курса) равенствам типа

17!=-288522240.

Неприятная неожиданность для человека, только приступающего к изучению программирования!

Еще одним важным фактором является простота методов, используемых в Ruby для ввода данных с клавиатуры или из файла. В случае языка Java сложность этих операций приходится тщательно маскировать, ограждая новичка от ненужных ему проблем.

Компактность дистрибутива Ruby и простота его установки в любой операционной системе позволяют школьнику или студенту без проблем работать на домашнем компьютере.

Язык Python по многим параметрам похож на Ruby. Из приведенных выше различий этих двух языков для рассматриваемой нами ситуации начального обучения программированию особенно значимым оказывается <<двумерный>> синтаксис, используемый в языке Python. Эта особенность не является изначально понятной и на первых порах мешает. Вообще, основным аргументом в пользу Ruby в качестве первого языка программирования является именно его интуитивная ясность и предсказуемость, а не многие другие его преимущества.