Eloquent Ruby by Russ Olsen,

「明解！Ruby」，ラス・オルセン著，(ピアソン桐原,2012).

第1章

Rubyのようにコードを書く

数年前、私は、大規模な文書管理システムの開発に長期間取り組んでいました。私はこの作業を通じて、プログラミングとは、正しく機能するコードを書くことはもちろんのこと、コミュニケーションが大事だということを実感しました。この開発は共同プロジェクトで、開発現場もアメリカの私のグループと東京に分散していました。開発者はそれぞれの国で開発を進め、会議で進捗の報告をしたり、プログラムをすりあわせたりします。しかし、会議は言語の壁にしばしば頓挫しました。そのうち、私が担当者同士の通訳にかり出されるようになり、本業のプログラミングはそっちのけで、半日以上もアメリカのスミス氏と東京の細川氏の間を行ったり来たりするというありさまでした。私が日本語を理解できるからとお思いですか？　とんでもない、その能力はまったくありません。ところが私には特殊能力[1]があるらしく、日本の同僚の教科書通りのカチカチの英語もスラングだらけのアメリカの同僚の米語もどちらも理解できます。それがどうやら彼らにはできないらしいのです。

同じようなことはプログラミング言語を使用するときに起こります。どんなプログラミング言語でも、そのパーサーも、有効なプログラムはすべて受け入れます。つまり、受け入れられれば、有効なプログラムとみなされるわけです。しかし、プログラミングコミュニティは、決まった記述形式、つまり共通の慣用的なコード（イディオム）を使用します。慣用的なコードを知れば、パーサーが何を受け入れるかを理解できます。これは、プログラミング言語を学ぶ上で非常に重要です。

この章で私たちは、最も少ないRubyのコードを使用して良いプログラムを書く冒険に出発します。Rubyコードをどのようにフォーマットしたら良いでしょうか？　Rubyコミュニティ（パーサーではなく）が変数名とメソッド名とクラス名に適用してきたルールは何でしょうか？　Rubyプログラムが文体的に正しいとされる理由を検証すると、Rubyプログラミングスタイルの背後にある考え方、Rubyを愛らしい、そして表現の豊かなプログラミング言語にしている真髄に迫ることができます。それでは出発しましょう。

いろはのい

原則としてRubyのプログラミングスタイルは、非常にシンプルな2つの考え方です。第1は、明確であること。良いコードは、コードが何をしようとしているか読者に正確に伝えます。第1の考えと関連しますが、良いコードとは単に明確であるだけでなく、簡潔であることが第2の考え方です。人間の脳が瞬間に理解できる情報量は限られています。一目ですべてを把握できれば、メソッドやクラスが何をするか理解するのは非常に簡単です。

実際に次のコードで、それを体感してみましょう。

1）私の特殊能力は数え切れない回数のランチとハッピーアワーズ、そして深夜の雑談のたまものです。それにどれだけ自分を捧げたことでしょう。

```
class Document
  attr_accessor :title, :author, :content

  def initialize( title, author, content )
    @title = title
    @author = author
    @content = content
  end

  def words
    @content.split
  end

  def word_count
    words.size
  end
end
```

Documentクラスは、技術的に特別なことは何もありません。著者のための1つのフィールド、タイトルのための1つのフィールド、内容のための1つのフィールドといくつかの単純なメソッドがあるだけです[2)]。上のコードで注目する点は、Rubyのインデント規約に従っていることです。Rubyでは、2個のスペースがインデントの最低単位です。スペース2個でレベル付けします。つまり、インデントの最初のレベルは2個、次は4個、その次は6個、その次は8個となります。2個のスペースというルールは、単なる思いつきにも見えますが、実際にはこれほど単純で、しかも実用的なルールはありません。1個では見分けにくいし、ワード間のスペースと区別がつきません。2個であれば何かの意図を感じさせます。3個では多すぎて、右側に情報のスペースが少なくなります。

Rubyの規約では、インデントに絶対にタブは使用しません。2個のスペースと同様に、タブの禁止も理由がはっきりしています。タブをスペースに変えるときに、人によってスペースの個数が違うからです。スペースの個数がまちまちだったり、タブと混在しているとコードが読みにくくなります。

```
class Document
  attr_accessor :title, :author, :content

  def initialize( title, author, content )
    @title = title
    @author = author
    @content = content
  end

  def words
    @content.split
  end

  def word_count
```

2) Documentクラスはこの本のあちこちで使用します。覚えておいてください。

```
      words.size
    end
  end
```

こんな些末なことで[3]で時間をむだにするには、人生はあまりにも短い。スケジュールもしかりです。タブなしのRubyこそがRubyというものです。

不要なコメントは控える

Rubyコメントは非常にシンプルです。#で開始するテキストがコメントです[4]。コメントはどこに、どれくらい挿入すればよいのでしょう。コメントはたくさん書いた方が親切でしょうか？

コードが主体となっているのが良いRubyの形です。つまり、コード自身に語らせるべきです。メソッドの使い方が誰にでも明らかで、クラスやプログラムで説明する必要がない場合、わざわざコメントを書かないでください。いつもそこにコメントを挿入するからという単純な理由で、コメントを書いてはいけません。決まり文句のようなコメントは控えましょう。

コードに挿入するコメントには正当な理由があります。ソフトウェアの使い方を説明することが第1の目的です。作成理由、使用しているアルゴリズム、速度をアップする方法などは不要です。明確に使い方だけを伝えてください。次の例は誰もが喜んで受け入れるでしょう。

```
# 書名と著者名の入ったプレーンテキストのドキュメントのモデルとなるクラス：
#
# doc = Document.new( 'Hamlet', 'Shakespeare', 'To be or...' ) )
# puts doc.title
# puts doc.author
# puts doc.content
#
# Document インスタンスは、そのコンテンツを単語に解析する方法を知っています：
#
# puts doc.words
# puts doc.word_count
#
class Document
  # クラスを省略・・・
end
```

コードの背景を説明するコメントを書くべきでないと言っているわけではありません。次に示すように、背景と指示を明確に分けてください。

3)「些末なこと」とは、タブとスペースが勝手気ままに混在した場合、フォーマットはカオスと化します。

4) Rubyは=beginと=endで区切られた複数行のコメントもサポートしていますが、Rubyのプログラマーは、多くの場合、#スタイルのコメントにこだわる傾向があります。

```
# 著者：ラス・オルセン
# Copyright 2010 Russ Olsen
#
# ドキュメント：Rubyクラスの例
```

特にコードが複雑な場合には「どのように動作するか」という説明を含めたほうが良い場合もあります。この場合も、その説明と使い方は分けるようにしてください。

```
# ngram分析を使用して、このドキュメントと渡された
# ドキュメントが同じ人によって書かれた確率を計算します。
# このアルゴリズムは、アメリカ英語に対して有効であることは
# 知られており、おそらくイギリス英語とカナダ英語でも機能し
# ます。
#
def same_author_probability( other_document )
  # 実装は読者の演習として残しておきます・・・
end
```

インラインコメントが役立つ場合もあります。

```
return 0 if divisor == 0  # ゼロによる除算を回避
```

この場合、コードの末尾にながながとコードの動作を説明するコメントを散りばめないでください。散りばめて良いのは華やかなクリスマスのイルミネーションだけです。

```
count += 1  # カウントを1加算
```

私が、コードの動作方法を説明するコメントを戒めるのは、できの悪いプログラムほどコメントで補うことが多いからです。プログラムは明快で簡潔でなければなりません。「コメントを追加する必要があるぞ」というささやきが聞こえたら、それは書き直しを求めるプログラムの叫びかもしれません。クラス名、メソッド名、変数名を変更して、コード自体が何をしようとしているか伝えるようにできませんか？　長いメソッドを分割したり、2つのクラスを1つにまとめたり、再構成できませんか？　アルゴリズムを再考できませんか？　サブタイトルなしでもコードだけでわかるようにできませんか？

良いコードは良いジョークのようなものという、古いプログラミングの格言を思い出してください。これは説明するまでもないでしょう。

クラスはキャメル、他はすべてスネーク

インデントとコメントという比較的簡単な問題を見てきましたが、ここで「名前」という問題に突き当たります。名前と言っても変数、クラス、メソッドを記述する単語についてではなく、これらの単語をどう組み合わせるかについての話しです。ルールは非常にシンプルです。いくつかの例外はありますが、メソッド、引数、インスタンス変数を含む変数は、lowercase_words_

separated_by_underscores(アンダースコアで区切った小文字の単語)を使用します[5]。

```
def count_words_in( the_string )
  the_words = the_string.split
  the_words.size
end
```

クラス名の命名規則はキャメルケースなので、DocumentやLegalDocumentはOKですが、Movie_scriptはNGです。このルールが実用的でないと感じるのであれば、定数を試してみてください。Rubyプログラマーの間では、定数をクラスのようにキャメルケースで表すかどうか意見が分かれているように見えます。

```
FurlongsPerFortnight = 0.0001663
```

あるいは、アンダースコアで区切ってすべて大文字にします。

```
ANTLERS_PER_MALE_MOOSE = 2
```

私はどちらかというと、本書のいたるところで記述されているように、定数のALL_UPPERCASE(すべて大文字)が好きです。

かっこは任意、あるいは禁止

Rubyでは、あってもなくても動作に影響しないものは構文で必須としていません。その代表がかっこです。メソッドの定義または呼び出しで、引数の前後にかっこを追加するか省略するかは自由です。したがって、ドキュメントを検索するメソッドを書く場合、かっこを使用すれば次のようになります。

```
def find_document( title, author )
  # 詳細は省略・・・
end

# ...

find_document( 'Frankenstein', 'Shelley' )
```

あるいは、次のようにかっこなしで書くことができます。

```
def find_document title, author
  # 詳細は省略・・・
end
```

5) この小文字の1つながりの様子から、この命名規則は「スネークケース」と呼ばれます。

```
# ...

find_document 'Frankenstein', 'Shelley'
```

いくつかの例外はありますが、Rubyプログラマーは一般にかっこの使用に賛成しています。多くのRubyプログラマーは、メソッドの定義でも呼び出しでも、メソッドに渡すものの前後をかっこで囲みます。これはかっこがあると、コードがわかりやすくなるように見えるためで、決して厳格なルールではありません。

もう一度言いますが、これは厳格な決まりではありません。Rubyプログラマーは、1行になじみのあるメソッド名が1つだけで、しかも引数の数が少ない場合、そのメソッドの呼び出しにはかっこを省略する傾向にあります。おなじみのputsはまさにこの例にぴったりです。一般的にputsの呼び出しではかっこを省略します。

```
puts 'Look Ma, no parentheses!'
```

"かっこの使用に賛成"に投票するルールに対するもう1つの例外は、引数リストを空にしないことです。引数のないメソッドを定義または呼び出す場合、かっこを省略して次のように書きます。

```
def words
  @content.split
end
```

次のようには書きません。

```
def words()
  @content.split()
end
```

最後に、制御文の条件にもかっこは不要です。省略するのが一般的であることを覚えておいてください。したがって、次のようには書きません。

```
if ( words.size < 100 )
  puts 'The document is not very long.'
end
```

一般的に次のように書きます。

```
if words.size < 100
  puts 'The document is not very long.'
end
```

行の畳み込み

多くのRubyコードは「1行に1文」形式にこだわりますが、複数の文を1行に詰め込んでもかまいません。その場合、文の間にセミコロンを挿入するだけです。

```
puts doc.title; puts doc.author
```

通常、上のように書くことはありませんが、多少の例外があります。たとえば、中身のないシンプルなクラスを定義する場合は、その定義を1行に畳み込みます。

```
class DocumentException < Exception; end
```

メソッドを使うこともできます。

```
def method_to_be_overriden; end
```

このような些細なことが非常に役立つことを覚えておいてください。目標は、明確で簡潔なコードを書くことです。大量の文を詰め込めば詰め込むほど、読みにくさが倍増します。

コードブロックの畳み込み

Rubyプログラムは、次のように一対の中かっこで区切られたコード群、すなわちコードブロックが多く使用されます。

```
10.times { |n| puts "The number is #{n}" }
```

または、キーワードdoとendで区切る場合もあります。

```
10.times do |n|
  puts "The number is #{n}"
  puts "Twice the number is #{n*2}"
end
```

この2つの形式のコードブロックは本質的に同じです。実際、どちらを使用してもかまいません。しかし、Rubyプログラマーはコードブロックの形式についてシンプルなルールを持っています。ブロックが単一の文で構成されている場合は、文全体を1行に畳み込み、中かっこで囲みます。一方、ブロックが複数の文で構成される場合は、行を分けdo/endの形式を使用します。

トラブルの回避

何よりも大切なことは、Rubyのようなコードは読みやすいということです。つまり、Rubyプログラマーは一般にこの章で説明したようなコーディングルールに従いますが、ときにはそれにとらわれずに、読みやすさを優先することが求められます。コードブロックを1行に畳み込むルールを守り、次のようには書きません。

```
doc.words.each do |word|
  puts word
end
```

おそらく次のように書くでしょう。

```
doc.words.each { |word| puts word }
```

1行のコードが長すぎる場合は、畳み込みのルールには従いません。

```
doc.words.each { |word| some_really_really_long_expression( ... with lots
of args ... ) }
```

どれくらいの長さを長すぎると判断するかは人により異なりますが[6]、多かれ少なかれいずれは、決断するときが来ます。

この決断にはかっこの使用も考慮すべきです。「ルール」ではかっこを省略する場合でも、残した方が読みやすい場合があります。たとえば、putsは一般にかっこを使用しないと説明しました。

```
puts doc.author
```

また、かっこを使用しない他の例としてinstance_of?があります。このメソッドは、引数のオブジェクトがクラスのインスタンスであるかどうかを判断します。

```
doc.instance_of? Document
```

しかし、次のような複雑な式にこの2つのメソッドを使用する場合はどうでしょうか。

```
puts doc.instance_of? self.class.superclass.class
```

この場合、読みやすさを考慮してかっこを使用するのがよいでしょう。

```
puts doc.instance_of?( self.class.superclass.class )
```

つまり、コードフォーマットのルールとは最終的に、現実的でそれぞれのケースに応じて、個別に判断することになります。

実践

何と言っても、慣用的なRubyコードの書き方を学ぶ最も良い方法は、慣用的なRubyコードを読むことです。Rubyインタープリタに同梱されているRuby標準ライブラリは、Rubyコードの集

6) 実際のところ、本書ではかなりの数のブロックを1行ではなく複数行で表していますが、これは単に長い行がページレイアウトからはみ出してしまうからです。

まりであり、最高の出発点です。関心のあるクラスを1つ取り上げてみてください[7]。おそらくSetクラスは気になる存在だと思います。Rubyインストールでset.rbを探し、興味を引く箇所に目を通してください。

set.rbファイルを開くと、2個のスペースのインデントや適格な変数名と共に、クラスの背景を説明するいくつかのコメント[8]から始まっていることがわかります。

```
# Copyright (c) 2002-2008 Akinori MUSHA <knu@iDaemons.org>
#
# Akinori MUSHAおよびGavin Sinclairによるドキュメンテーション。
#
# # 著作権所有。Rubyと同じ条件のもとで再配布・修正すること
# ができます。
```

この後に、このクラスの短い動作説明が続きます。

```
# このライブラリは、重複のない順不同の値のコレクションを処理する
# Setクラスを提供します。これは、配列の直感的相互操作機能とハッシュの
# 高速検索の混成です。値の順序を保持する必要がある場合は、SortedSet
# クラスを使用してください。
```

ここでいくつかの例を示します。

```
# require 'set'
# s1 = Set.new [1, 2]                   # -> #<Set: {1, 2}>
# s2 = [1, 2].to_set                    # -> #<Set: {1, 2}>
# s1 == s2                              # -> true
```

コメントとして役に立つものを追加しようとして、set.rbのひな形に従ことはおろか、より悪いことをしてしまうかもしれません。

Setクラスをよく見ると、追加されたいくつかのメソッド名のルールが使用されています。まずは疑問符です。通常、Rubyプログラマーはyes/noまたはtrue/falseを返すメソッド名の後ろに疑問符を付けます。確かにSet内には、include?、superset?、empty?などのメソッドが確認できます。でも疑問符の姿にまどわされてはいけません。これは単にメソッド名の一部にすぎず、特別なRuby構文ではありません。感嘆符も同様にRubyのルールでは、「!」はメソッド名の末尾に付けてもよい文字です。Rubyのプログラマーが「!」を使用するときは、何か予期しないこと、場合によっては危険な結果になることを意味しています。Setクラスにはflatten!とmap!メソッドがありますが、いずれも変更されたコピーを返す代わりに、クラスそのものを変更してしまいます。

set.rbはRubyの作法のひな形ですが、組み込みコードの中にはRubyのルール違反をしているものがあります。代表的な例が、Floatメソッドです。大文字で始まるメソッド名です。ショッ

7) もっと頻繁に取り出すべきでしょう。
8) 実際に見てみれば、コメントの形式をページレイアウトに合わせ、少し修正してあることに気づくはずです。

クです！[チャララ〜チャラララッ]

この重大なマナー違反は少し弁護する必要があります。Floatメソッドは、その引数（通常は文字列）を浮動小数点数に変換します[9]。したがって、Floatは、クラス名の代わりに使用できます。

```
pi = Float('3.14159')
```

ルールのルールたるゆえんは、必ず誰かが破るということです。そして、その破る誰かが関係当局だったりすると、なおさらおもしろくなります。

まとめ

スペースによる最小単位のインデント。名前の付け方の明確なルール。通常は付けられるけれども、付けなくてもよいかっこ。使用方法、動作説明、著者について、それぞれ控えめなコメント。何にもまして、現実主義。以上がRubyプログラミングスタイルのまさしく「いろはのい」です。ルールに盲目的に従うだけでは、コードが読みやすくなるわけがありません。

しかしこれは始まりにすぎません。もっとたくさんのRubyのルールを本書全体で紹介します。特に、第10章と第18章ではメソッド名の問題に戻ります。最後の章はおどろくなかれルール違反のすすめです。

しかし、しばらくは基本に沿って進みます。次の章では、Rubyの制御構造を説明し、それがコードを明確で簡潔にするために、どのように役立っているかを見ていきます。

9) Float('3.14159')は'3.14159'.to_fと同じではありません。Floatメソッドは、間違った入力を渡すと例外を返しますが、to_fは黙って0を返します。