

B.3.4 環境変数 .....	482
あとがき .....	483
謝辞 .....	484
索引 .....	485

「だって、じゃあ、考えないで何かやっちゃうの？」

—— 新井素子「…… 絶句」

## 第1部

# Rubyを はじめよう

簡単なプログラムを見ながら  
Rubyについての全体像を  
眺めていきましょう。  
Rubyを使ったプログラミングの  
イメージをつかんでください。

# はじめてのRuby

それではRubyを使ってみましょう。

この章では次の内容について紹介します。Rubyを使ったプログラミングの概要をつかんでください。

- Rubyを使う

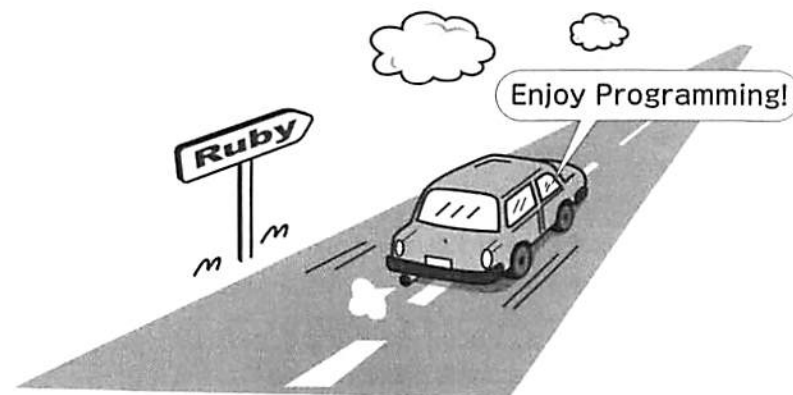
Rubyを使って、プログラムを実行してみます。

- 文字や数値を使う

文字や数値を出力したり、計算を行ったり、変数に代入したりします。

- 条件判断や繰り返しを行う

数値や文字列を比較して条件判断や、処理の繰り返しを行います。




## 1.1 Rubyの実行方法

まずは、画面に文字を表示するプログラムを作り、実行してみましょう。

Rubyで書かれたプログラムを実行する方法はいくつかあります。いちばん一般的なのは、`ruby`というコマンドを使って実行する方法です。次によく使うのが、`irb`というコマンドを使って対話的に実行する方法です。小さいRubyプログラムのときは、`irb`コマンドを使うほうが簡単に実行できます。

ここではまず`ruby`コマンドを使う方法を紹介して、その後で`irb`コマンドを使う方法も紹介します。

なお、Rubyそのものをインストールしていない人は、「付録A Ruby実行環境の準備」を見て、あらかじめインストールしておいてください。


 本書で使用するRubyのバージョンは、Ruby 2.3です。Mac OS XやLinuxを使用している場合、標準でインストールされているRubyのバージョンが古いことがあります。その場合、新しいRubyをインストールしてください。

### 1.1.1 rubyコマンドを使う方法

では、はじめて実行するプログラムList 1.1を見てみましょう。


List 1.1 `helloruby.rb`

```
print("Hello, Ruby.\n")
```


 「\」(バックスラッシュ)は、Windowsでは「¥」(円記号)と表示されます。本書の中では、原則として「\」に統一します。

……ちょっと拍子抜けしたでしょうか? 「プログラム」と聞いて、何かすごく長い暗号めいたものを想像されたかもしれませんが、このプログラムはたったの1行です。文字数にしても20字ちょっとしかありません。でも、これも立派なプログラムですし、実行すればちゃんと目的を果たします。

このプログラムをエディタで入力し、ファイル名を「`helloruby.rb`」にして、ファイルとして保存してください。ファイル名の「`.rb`」は、Rubyのプログラムであることを表しています。

 プログラムを入力するには、「エディタ」または「IDE」というソフトウェアを使います。エディタやIDEについては、「A.5 エディタとIDE」(p.471)をご覧ください。

それでは、このプログラムを実行してみましょう。コンソールを起動します。

 コンソールの起動方法については、「付録A Ruby実行環境の準備」(p.461)でOS別に説明しています。

コンソールを起動したら、ファイル`helloruby.rb`を置いたフォルダに、`cd`コマンドで移動します。たとえばWindowsを使っていて、Cドライブの`src`フォルダ(`c:\src`)にファイルを置いたのであれば、次のように入力します。

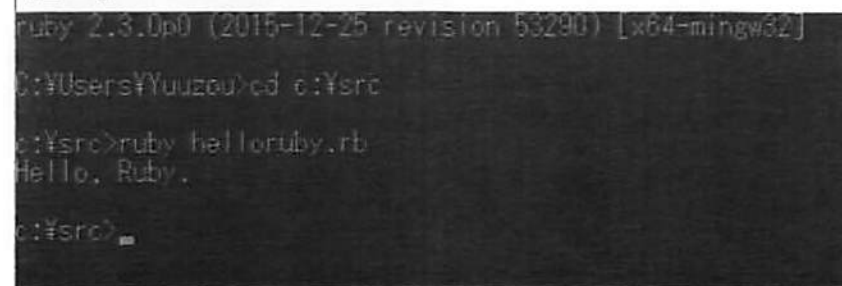
```
> cd c:\src
```

そこで、

```
> ruby helloruby.rb
```


と入力します。すると、図1.1のように「Hello, Ruby.」と表示されます。

図 1.1 Ruby コマンドプロンプトを開く



```
ruby 2.3.0p0 (2015-12-25 revision 53290) [x64-mingw32]
C:\Users\Yuuzeu>cd c:\src
C:\src>ruby helloruby.rb
Hello, Ruby.
C:\src>
```

図 1.1 rubyの起動

 もしエラーが出てしまうようなら、第10章のコラム「エラーメッセージ」(p.193)および「付録A Ruby実行環境の準備」(p.461)を確認してください。

### 1.1.2 irbコマンドを使う方法

irbコマンドを使う方法も紹介します。

irbコマンドは、rubyコマンドと同様にコンソールから実行します。ただし、プログラムを書いたファイルは指定しません。

irbコマンドを実行すると、次のように入力プロンプトが表示されます。

#### 実行例

```
> irb
irb(main):001:0>
```

ここで、先ほどのプログラムList 1.1をそのまま入力し、[Enter]キーを押すと、その場で実行されます。

#### 実行例

```
irb(main):001:0> print("Hello, Ruby.\n")
Hello, Ruby. ← printメソッドによって表示された文字列
=> nil
irb(main):002:0>
```

**メモ** 3行目に表示される「=> nil」というのは、print文自体の戻り値です。詳しくは「7.3.1 メソッドの戻り値」(p.108)で説明します。

このように、入力したプログラムをその場で実行できるので、簡単なテストにはとても便利です。ただし、大きなプログラムを試すのには不向きなので、そのような場合にはrubyコマンドを使いましょう。

irbコマンドを終了するには「exit」と入力して[Enter]キーを押すか、[Control]([Ctrl])キーを押しながら[d]キーを押します。

**注意** Mac OS XやWindowsを使っている場合、irbコマンドでは日本語が正しく入力できないことがあります。その場合、irbコマンドに--noreadlineオプションをつけて「irb --noreadline」と実行してください。これでreadline機能がオフになり、日本語を正しく入力できるようになります。ただし、readline機能をオフにすると、入力済みの文字の編集機能やヒストリ入力支援機能などが使えなくなってしまうので注意してください。

## 1.2 プログラムの解説

それでは、ほんの1行だけではありますが、List 1.1のプログラムを解説しましょう。

### 1.2.1 オブジェクト

まず、「"Hello, Ruby.\n"」という部分に注目します。

```
print("Hello, Ruby.\n")
      └── 文字列オブジェクト
```

これをStringオブジェクト、または文字列オブジェクト、あるいは単に文字列と呼びます。「Hello, Ruby.」という文字列を意味するオブジェクト、というわけです(図1.2)。

文字列オブジェクト  
「Hello, Ruby.」 → "Hello, Ruby.\n"

データは、プログラム中ではオブジェクトとして表現される

#### 図1.2 データとオブジェクト

Rubyでは、文字列、数値、時刻などさまざまなデータがオブジェクトになります。

**メモ** 文字列の終わりの「\n」は改行を表す文字です。

### 1.2.2 メソッド

今度は「print」という部分に注目しましょう。

```
print("Hello, Ruby.\n")
      └── メソッド └── 引数
```

「print」は、メソッドです。メソッドとは、オブジェクトを扱うための手続きのことです。「数値」を使って足し算や掛け算をしたり、「文字列」同士をつなげたり、「ある時刻」の1時間後や1日後を求めたりといったことは、すべてメソッドを起動することによって行われます。

printメソッドは、「()」の中の内容をコンソールに出力するメソッドです。ですから、helloruby.rbでは、「Hello, Ruby.」という文字列オブジェクトが表示されています。

メソッドに渡す情報のことを<sup>ひきすう</sup>引数といいます。たとえば、printメソッドの機能を説明する場合に「printメソッドは引数として与えられた文字列をコンソールに出力します」といった使い方をします。

printメソッドの引数を書き換えて、別の文字列を表示するプログラムにしてみましょう。

```
print("Hello, RUBY!\n")
```

今度は大文字で「Hello, RUBY!」と表示するようになります。ちょっと元気のよいあいさつになりましたか？

## 1.3 文字列

文字列について、もう少し詳しく見ていくことにしましょう。

### 1.3.1 改行文字と「\n」

先ほど、文字列の「\n」は改行を表すと説明しました。普通の文字を使って改行を書けるおかげで、たとえば

```
Hello,
Ruby
!
```

と表示させるには、

```
print("Hello, \nRuby\n!\n")
```

↑ ↑ ↑ 改行文字

と書くことができます。もっとも、

```
print("Hello,
Ruby
!")
```

など書いても、同じように表示されます。しかし、この書き方だとプログラムが読みにくくなってしまいますので、あまりよい書き方ではありません。せっかく改行を表す書き方があるので、それを使うほうがよいでしょう。

「\n」以外にも、文字列の中で特殊な文字を埋め込みたいときに「\」を使います。たとえば、「"」は文字列の始まりと終わりを表す文字ですが、これを文字列の中を含める場合には「\"」とします。

```
print("Hello, \"Ruby\".\n")
```

は、

```
Hello, "Ruby".
```

と表示されます。

このように、文字列中の「\」はそれに続く文字に特別な意味を与える文字になっています。そのため、「\」そのものを文字列中に含めたいときには、「\\」と書く必要があります。たとえば

```
print("Hello \\ Ruby!")
```

は、



```
Hello \ Ruby!
```

と表示されます。2つあった「\」が1つになっていることに注意してください。

### ● 1.3.2 「'」と「"」

文字列オブジェクトを作るための区切り文字には、「"」(ダブルクォート)ではなく、「'」(シングルクォート)を使うこともできます。先ほどのプログラムを

```
print('Hello, \nRuby\n!\n')
```

とシングルクォートに書き換えて実行してみましょう。すると今度は、

```
Hello, \nRuby\n!\n
```

というように、「'」の中の文字がそのまま表示されます。

このように「'」で囲った文字列は、「\n」などの特殊文字の解釈を行わず、そのまま表示します。ただし例外として、「\」と「'」を、文字列中に文字そのものとして含めたいときにのみ、その文字の前に「\」をつけます。こんな感じです。

```
print('Hello, \\ \'Ruby\'.')
```

実行すると次のように表示されます。

```
Hello, \ 'Ruby'.
```

## 1.4 メソッドの呼び出し

メソッドについてもう少し説明しましょう。

Rubyのメソッドでは「()」を省略することができます。そのため、先ほどのプログラム(List 1.1)でのprintメソッドは、

```
print "Hello, Ruby.\n"
```

と書くこともできます。

また、いくつかの文字列を続けて表示したいときには、「,」で区切れば、並べた順に表示できます。ですから

```
print "Hello, ", "Ruby", "!", "\n"
```

なんて書き方もできるわけですね。これは、表示したいものがいくつもあるときに使うと便利です。とはいえ、要素が複雑に混み入ってくると、「()」をつけたほうがわかりやすくなります。慣れるまではこまめに「()」を書いておきましょう。本書では、単純な場合には「()」を省いて表記しています。

さらに、メソッドを縦に並べて書くと、その順にメソッドを実行します。たとえば

```
print "Hello, "
print "Ruby"
print "."
print "\n"
```

など書いても、同じように「Hello, Ruby.」と表示するプログラムになります。

## 1.5 putsメソッド

printメソッド以外にも文字列を表示するメソッドがあります。putsメソッドは、printメソッドとは異なり、表示する文字列の最後で必ず改行します。これを使えば、List 1.1は

```
puts "Hello, Ruby."
```

と書けるようになります。ただし、

```
puts "Hello, ", "Ruby!"
```

のように2つの文字列を渡した場合には、

```
Hello,
Ruby!
```

と、それぞれの文字列の末尾に改行が追加されます。printメソッドとは少し使い勝手が違いますね。この2つのメソッドは、場面に応じて使い分けてください。

## 1.6 pメソッド

さらにもう1つ、表示のためのメソッドを紹介しましょう。オブジェクトの内容を表示するときに便利な「p」というメソッドです。

たとえば、数値の100と文字列の"100"を、printメソッドやputsメソッドで表示させると、どちらも単に「100」と表示されてしまいます。これでは本当はどちらのオブジェクトなのか、表示結果から確認できません。そんなときには、pメソッドを使うのが便利です。pメソッドなら、文字列と数値を違った形で表示してくれるのです。さっそく試してみましょう。

```
puts "100"  #=> 100
puts 100    #=> 100
p "100"     #=> "100"
p 100       #=> 100
```

**メモ** 本書では、プログラム中で出力した内容を表すために、出力用のメソッドの横に「#=>」という文字を置き、その右側に出力された文字を並べて書くという表記を用いています。この例では、「puts "100"」や「puts 100」、「p 100」というメソッドでは「100」という文字列が出力され、「p "100"」というメソッドでは「"100"」という文字列が出力される、という意味になります。

このように、文字列を出力する場合、「"」で囲んで表示してくれるわけです。これなら一目瞭然ですね。さらに、文字列の中に含まれる改行やタブなどの特殊な文字も、「\n」や「\t」のように表示されます (List 1.2)。

### List 1.2 puts\_and\_p.rb

```
puts "Hello, \n\tRuby."
p "Hello, \n\tRuby."
```

### 実行例

```
> ruby puts_and_p.rb
Hello,
    Ruby.
"Hello, \n\tRuby."
```

printメソッドは実行結果やメッセージなどを普通に表示したいとき、pメソッドは実行中のプログラムの様子を確認したいとき、と使い分けられよいでしょう。原則として、pメソッドはプログラムを書いている人のためのメソッドなのです。

## 1.7 日本語の表示

ここまで、文字列にはアルファベット（英字）を使ってきました。

今度は日本語を表示してみましょう。日本語の表示も難しいことは何もありません。単にアルファベットの代わりに日本語を「」の中を書くだけです。こんな感じになります。

### List 1.3 kiritsubo.rb

```
print "いつれの御時にか女御更衣あまたさぶらいたまいけるなかに\n"
print "いとやむごとなき際にはあらぬがすぐれて時めきたまふありけり\n"
```

#### 実行例

```
> ruby kiritsubo.rb
いつれの御時にか女御更衣あまたさぶらいたまいけるなかに
いとやむごとなき際にはあらぬがすぐれて時めきたまふありけり
```

ただし、文字コードの設定によっては、エラーが出たり、正しく表示されない場合があります。その場合、コラム「日本語を扱う場合の注意」を参照してください。

### Column

#### 日本語を扱う場合の注意

環境によっては、日本語を含むスクリプトを実行すると次のようなエラーになります。

#### 実行例

```
> ruby kiritsubo.rb
kiritsubo.rb:1: invalid multibyte char (UTF-8)
kiritsubo.rb:1: invalid multibyte char (UTF-8)
```

これはソースコードの文字コード（エンコーディング）が指定されていない

からです。Rubyでは、「**# encoding: 文字コード**」というコメントを1行目に記述することによってソースコードの文字コードを指定します（文字コードを決めるルールのことをエンコーディングといいます）。このコメントをマジックコメントといいます。

Windowsで一般的に使われているエンコーディングShift\_JISでソースコードを記述した場合は、次のようにマジックコメントを書きます。

```
# encoding: Shift_JIS
print "いつれの御時にか女御更衣あまたさぶらいたまいけるなかに\n"
print "いとやむごとなき際にはあらぬがすぐれて時めきたまふありけり\n"
```

このようにコメントで文字コードを指定することによって、Rubyがソースコード中の日本語を正しく認識できるようになります。次の表にプラットフォームごとによく使われる文字コードをまとめています。複数の文字コード名が挙げられている場合は、環境に合わせて適切なものを選んでください。

プラットフォーム	文字コード（エンコーディング）名
Windows	Shift_JIS（またはWindows-31J）
Mac OS X	UTF-8
Unix	UTF-8、EUC-JPなど

なお、マジックコメントがないソースコードの文字コードはUTF-8と仮定されます。そのため、UTF-8のソースコードを使う場合はマジックコメントは不要です。

これ以外でも、前述のpメソッドで日本語の文字列を出力すると、いわゆる「文字化け」をしたような出力になる場合があります。そのような場合は、出力用の文字コードを指定するために「**-E 文字コード**」の形式でコマンドラインオプションを指定してください。コンソールがUTF-8を受けつける場合は次のようになります。

#### 実行例

```
> ruby -E UTF-8 スクリプトファイル名 ← スクリプトの実行
> irb -E UTF-8 ← irbの起動
```



## 1.8 数値の表示と計算

文字列に続いて、今度は「数値」を扱ってみましょう。Rubyのプログラムでは、整数や小数（浮動小数点数）を、自然な形で扱うことができます。

### 1.8.1 数値の表示

まずは文字列の代わりに数値を表示するところから始めてみます。「1.2 プログラムの解説」(p.7)で、「Rubyでは文字列は文字列オブジェクトという形になっている」と説明しました。同じように、数値も「数値オブジェクト」として扱われます。

Rubyで整数オブジェクトを表現するのは簡単です。そのまま数字を書けばよいだけです。たとえば

```
1
```

と書けば「1」の値の整数 (Fixnum) オブジェクトになります。また、

```
100
```

と書けば、「100」の値の整数オブジェクトになります。

さらに、

```
3.1415
```

などと書けば、「3.1415」の値の浮動小数点数 (Float) オブジェクトになります。

**メモ** 「Fixnum」や「Float」というのは、それぞれのオブジェクトが所属する「クラス」の名前です。クラスについては、第4章と第8章で説明します。

数値を表示するには、文字列と同様にprintメソッドやputsメソッドを使います。

```
puts(10)
```

というメソッドを実行すると、

**実行例**

```
10
```

と画面に表示されます。

### 1.8.2 四則演算

数の計算を行ったり、その結果を表示したりすることもできます。四則演算をやってみましょう。

ここではirbコマンドを使ってみます。

**実行例**

```
> irb --simple-prompt
>> 1 + 1
=> 2 ← 1 + 1の実行結果
>> 2 - 3
=> -1 ← 2 - 3の実行結果
>> 5 * 10
=> 50 ← 5 * 10の実行結果
>> 100 / 4
=> 25 ← 100 / 4の実行結果
```

**メモ** irbコマンドのあとの--simple-promptは、irbのプロンプト表示を簡易にするためのオプションです。

プログラミング言語の世界では、掛け算の記号に「\*」（アスタリスク）を、割り算の記号に「/」（スラッシュ）を使うのが一般的です。Rubyもこの習慣になっています。

もう少し四則演算をやってみましょう。普通の計算では、「足し算・引き算」

と「掛け算・割り算」には計算の順序が決められていますが、Rubyでも同じです。つまり、

```
20 + 8 / 2
```

とすれば答えは「24」になります。「20 + 8」を2で割りたいときは、「()」で囲って、

```
(20 + 8) / 2
```

とします。答えは「14」になります。

### ● 1.8.3 数学的な関数

四則演算以外にも、平方根や、三角関数の「sin」「cos」、指数関数などの数学的な関数が利用できます。ただし、その場合、関数の前に「Math.」という文字列をつける必要があります。

**メモ** 「Math.」をつけずに「sin」「cos」などの関数を使うには、「include Math」という文が必要です。これについては「8.7.2 名前空間の提供」で説明します。

sinはsinメソッド、平方根はsqrtメソッドで求めます。メソッドを実行すると、計算した結果を得ることができます。このことを「メソッドが値を返す」といい、得られる値のことを戻り値といいます。

#### 実行例

```
> irb --simple-prompt
>> Math.sin(3.1415)
=> 9.26535896604902e-05 ← sinメソッドの戻り値
>> Math.sqrt(10000)
=> 100.0 ← sqrtメソッドの戻り値
```

**注意** Rubyのバージョンや実行する環境により、結果の桁数などが異なる場合があります。

1番目のsinの答えである「9.26535896604902e-05」ですが、これは、極端に大きい数や、極端に小さい数を表すときに使われる表記方法です。「(小数)e(整数)」と表示されたときは、「(小数) \* [10の(整数)乗]」の値、と解釈してください。この例の場合、「9.26535896604902 × 10<sup>-5</sup>」ということになるので、つまりは0.0000926535896604902という値を表しています。

## 1.9 変数

プログラミングに欠かせない要素として変数があります。変数とは、「もの」につける名札のようなものです。

オブジェクトに名札をつけるには、

変数名 = オブジェクト

と書きます(図1.3)。このことを「変数にオブジェクトを代入する」といいます。

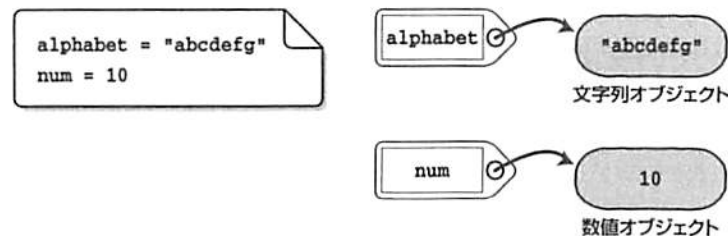


図 1.3 変数とオブジェクト

```
alphabet = "abcdefg"
num = 10
age = 18
name = 'TAKAHASHI'
```

変数の利用例として、直方体の表面積と体積を求めるプログラム(List 1.4)を見てみましょう。

**List 1.4** area\_volume.rb

```
x = 10
y = 20
z = 30
area = (x*y + y*z + z*x) * 2
volume = x * y * z
print "表面積=", area, "\n"
print "体積=", volume, "\n"
```

変数をまったく使わなければ、

```
print "表面積=", (10*20 + 20*30 + 30*10) * 2, "\n"
print "体積=", 10*20*30, "\n"
```

といったプログラムになってしまいます。これでは値を1つ変更するために何カ所も修正しなければいけません。この例はたったの2行なのでたいしたことはありませんが、ちょっと大きなプログラムになると、そのような変更をきちんと行うのは大変な手間となります。

また、変数には、値が何を表しているのかを明確にするという意味もあります。したがって、わかりやすい名前をつけることが大切です。たとえば、

```
hoge = (foo*bar + bar*baz + baz*foo) * 2
funi = foo * bar * baz
```

という調子では、何をやっているのかさっぱりわからないプログラムになってしまいます。変数名には、「area」や「volume」など、そのまま意味のわかる単語などを使うように、ふだんから心掛けましょう。

### ● 1.9.1 printメソッドと変数

printメソッドの動きをもう少し見てみましょう。

```
print "表面積=", area, "\n"
```

このprintメソッドの呼び出しには「"表面積="」「area」「"\n"」の3つの引数を指定しています。printメソッドはこれらの引数の値を順番に出力します。

「"表面積="」は「表面積=」という値を持った文字列なので、それがそのまま出力されます。「area」はareaという変数に関連づけられたオブジェクトになります。この例では2200という整数になっているので、printメソッドはその値を出力します。

最後の「"\n"」は改行を表す文字列なので、そのまま出力します。

これらの3つの値をprintメソッドで処理した結果として「表面積=2200」と改行が画面に表示されるというわけです。

printメソッドに渡す文字列は次のように書くこともできます。

```
print "表面積=#{area}\n"
```

「"表面積=#{area}\n"」が全体で1つの文字列になっています。「#{area}」は文字列の中に変数areaの値を埋め込むという書き方です。文字列の中に「#{変数名}」と書くと、文字列にデータを埋め込むことができます。計算結果の変数名を埋め込む代わりに、「"表面積=#{(x\*y + y\*z + z\*x) \* 2}\n"」のように計算式を直接書いても同じ結果を得られます。

画面に結果を出力する場合は改行も出力することが多いため、putsメソッドを使って次のように書けば、「\n」も必要なくなり、プログラムがすっきりします。

```
puts "表面積=#{area}"
```

## 1.10 コメントを書く

プログラムの中には、コメントを書くことができます。コメントは、プログラム中に書かれていても、直接プログラムとしては扱われません。つまり、プログラムの実行には何の関係もないもの、ということです。

「どうしてプログラムの中に、実行とは関係のない余計なものを書くのだから？」と思われるかもしれませんが。確かに一度書いて実行すればそれっきり、というプログラムであれば、コメントは特に必要ないでしょう。しかし、一度書いたプログラムを何度も使いまわすことも少なくありません。そのようなときに、

- プログラムの名前や作者、配布条件などの情報
- プログラムの説明

などを書いておくために、コメントが使われます。

コメントを表す記号は「#」です。行頭に「#」があれば、1行まるまるコメントになります。行の途中に「#」があれば、「#」の部分から行末までがすべてコメントになります。また、行頭から始まる「=begin」と「=end」で囲まれた部分もコメントになります。これは、プログラムの先頭や最後で、長い説明を記しておくのに重宝します。

List 1.5は、先ほどのList 1.4にコメントを追加したプログラムです。濃い網掛けの部分がコメントになっています。

List 1.5 comment\_sample.rb

```
=begin
「たのしいRuby 第5版」サンプル
コメントの使い方の例
2006/06/16 作成
2006/07/01 一部コメントを追加
2015/10/01 第5版用に更新
=end
x = 10 # 横
y = 20 # 縦
z = 30 # 高さ
# 表面積と体積を計算する
area = (x*y + y*z + z*x) * 2
volume = x * y * z
# 出力する
print "表面積=", area, "\n"
print "体積=", volume, "\n"
```

なお、コメントは、先ほど挙げた目的以外にも、「この行の処理を一時的に実行させないようにする」といったことにも使います。

C言語のコメントのように、行の途中だけをコメントにするような書き方はありません。行末まで必ずコメントになります。

## 1.11 制御構造

プログラミング言語には、制御構造というものがあります。

これは、何かの条件によって、プログラムの実行順序を変えたり、プログラムの一部を実行させなかったりするための仕掛けです。

### 1.11.1 制御構造の分類

制御構造を大雑把に分類すると次のようになります。

- 逐次処理：プログラムを書かれた通りに、先頭から順に実行する
- 条件判断：ある条件が成り立つ場合は〇〇を、そうでない場合は××を実行する
- 繰り返し：ある条件が成り立つ間、〇〇を繰り返し実行する
- 例外処理：何か例外が発生した場合には、〇〇を実行する

逐次処理というのは、通常の処理のことです。特に何も指定していない場合、プログラムは書かれた順に実行されます。

条件判断は、条件に応じて処理が分岐します。条件が満たされない場合、書かれた処理の一部を飛ばして、実行が行われます。Rubyでは、if文やunless文、case文などが条件判断文になります。

繰り返しは、条件に応じて、ある処理を何度も繰り返して実行することです。この場合、書かれた順序に逆らって、すでに一度実行されているところに戻って、再度実行が行われます。

例外処理はやや特殊です。想定していない問題が発生したとき、それまで実行していた部分を抜け出して、別の場所から実行を再開する処理です。場合によっては、そこでプログラムが終了してしまうこともあります。

ここでは「条件判断」と「繰り返し」を取りあげます。



## 1.12 条件判断:if ~ then ~ end

ある条件によって挙動が変わるプログラムを作るには、if文を使います。if文の構文は、次のようになります。

```
if 条件 then
  条件が成り立ったときに実行したい処理
end
```

条件には、値がtrueまたはfalseとなる式を書くのが一般的です。2つの値を比較して、一致すればtrue、一致しなければfalse、などが条件にあたります。

数値の場合、たとえば大小関係の比較には、等号や不等号を使います。Rubyでは、「=」は代入のための記号として使われるので、一致するかどうか調べるには「==」を2つ並べた記号「==」を使います。また、「≤」と「≥」には、「<=」と「>=」を使います。

このような比較の結果はtrueまたはfalseとなります。もちろん、trueはその条件が成り立っている場合、falseは成り立っていない場合です。

```
p (2 == 2) #=> true
p (1 == 2) #=> false
p (3 > 1)  #=> true
p (3 > 3)  #=> false
p (3 >= 3) #=> true
p (3 < 1)  #=> false
p (3 < 3)  #=> false
p (3 <= 3) #=> true
```

文字列の比較もできます。この場合も「==」を使います。同じ文字列ならtrue、異なる文字列ならfalseを返します。

```
p ("Ruby" == "Ruby") #=> true
p ("Ruby" == "Rubens") #=> false
```

値が異なっていることを判断するには、「!=」を使います。これは「≠」の意味ですね。

```
p ("Ruby" != "Rubens") #=> true
p (1 != 1)              #=> false
```

では、これらを使って、条件判断文を書いてみましょう。変数aの値が10以上の場合は「greater」、9以下の場合は「smaller」と表示するプログラムはList 1.6のようになります。

List 1.6 greater\_smaller.rb

```
a = 20
if a >= 10 then
  print "greater\n"
end
if a <= 9 then
  print "smaller\n"
end
```

thenは省略することもできます。その場合、if文は次のようになります。

```
if a >= 10
  print "greater\n"
end
:
```

また、条件に一致するときとしないときで違う動作をさせたい場合は、elseを使います。次のような構文になります。

```
if 条件 then
  条件が成り立ったときに実行したい処理
else
  条件が成り立たなかったときに実行したい処理
end
```

これを使って、List 1.6を書き直すと、List 1.7のようになります。



## List 1.7 greater\_smaller\_else.rb

```
a = 20
if a >= 10
  print "greater\n"
else
  print "smaller\n"
end
```

## 1.13 繰り返し

同じこと、または同じようなことを何度も繰り返したい場合があります。繰り返しを行う方法を2つ紹介しましょう。

### 1.13.1 while文

while文は、繰り返しを行うための基本的な構文です。なお、doは省略することもできます。

```
while 繰り返し続ける条件 do
  繰り返したい処理
end
```

#### ○ 例: 1から10までの数を順番に表示する

```
i = 1
while i <= 10
  print i, "\n"
  i = i + 1
end
```

### 1.13.2 timesメソッド

繰り返しの回数が決まっているときは、「times」というメソッドを使うとシンプルにできます。なお、こちらの「do」は省略できません。

```
繰り返す回数.times do
  繰り返したい処理
end
```

#### ○ 例: 「All work and no play makes Jack a dull boy.」と100行表示する

```
100.times do
  print "All work and no play makes Jack a dull boy.\n"
end
```

timesメソッドはイテレータと呼ばれるメソッドです。イテレータ(iterator)は、Rubyの特徴的な機能です。スペルからもわかるように「繰り返す(iterate)もの(-or)」という意味です。オペレータ(operator)が「演算(operate)するもの」として「演算子」と呼ばれるのを真似るなら、さしずめ「繰り返し子」「反復子」というところでしょうか。その名の通り、繰り返しを行うためのメソッドです。

Rubyはtimesメソッド以外にも数多くのイテレータを提供しています。イテレータの代表はeachメソッドです。eachメソッドについては、第2章で配列やハッシュと一緒に紹介します。