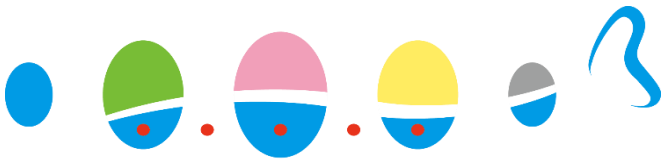
Rubyエデュケーション

Rubyプログラミング技術者育成エデュケーション



Rubyエデュケーション(目次)

● イントロダクション

●本セミナーの目標、方針について

◆ **目標**

プログラム言語Rubyの技術を修得する。
→セミナー受講後、公式Ruby技術者認定試験（Silver）に挑戦できるレベルを目指します。

◆ **セミナーの進め方**

全20回を予定。
5回ずつの4クールに分割してテーマを設定し、段階を追って理解度を深める。

1 環境導入～基本的なRubyプログラミング
2 Rubyの特徴的な技術を用いたプログラミング
3 クラスとオブジェクトの利用
4 生産効率を向上させるプログラミング

*1クール単位の進め方について

1回目	2回目	3回目	4回目	5回目
講習 実技	講習 実技	講習 実技	講習 実技	演習課題 筆記試験

1つのクールの終わり毎に、技術習熟度の確認として演習と試験を実施。

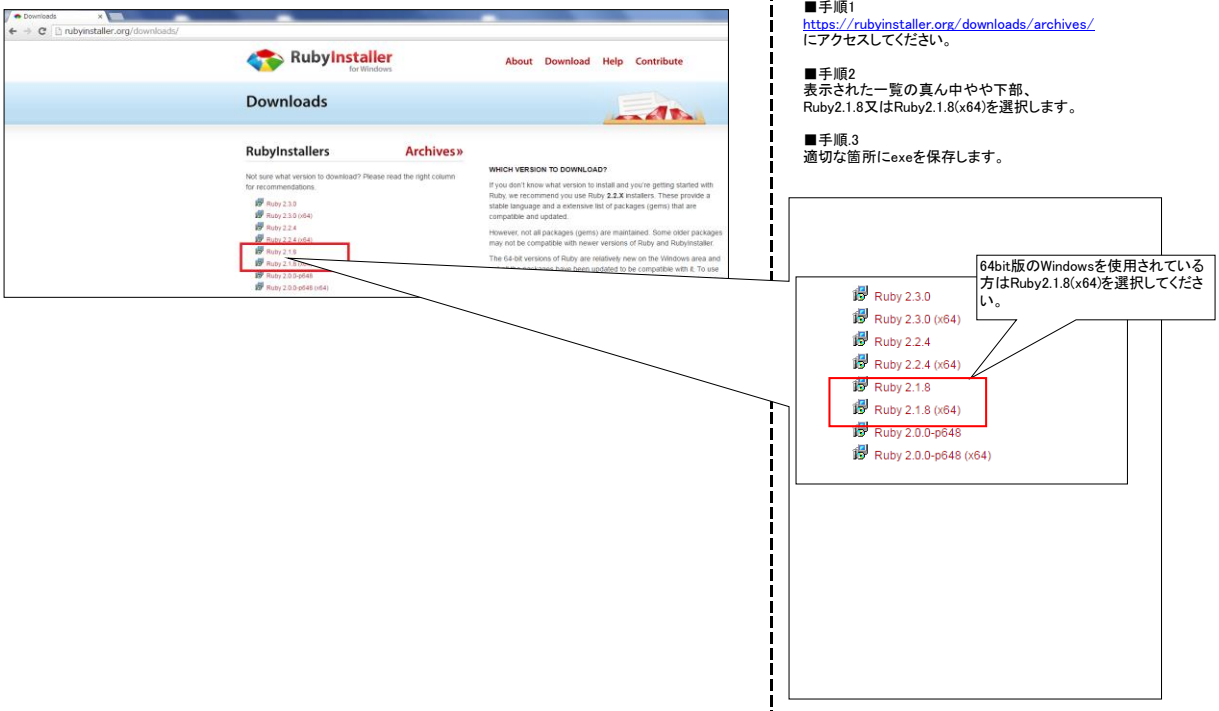
● 環境設定について

◆ ダウンロード
◆ インストール
◆ HelloWorld出力

環境設定については、次ページ「環境設定」をご覧ください。

●環境設定

■手順2 – Ruby2.1.8又はRuby2.1.8(x64)を選択



*次の手順に沿って、必要な環境を作ってゆきましょう。

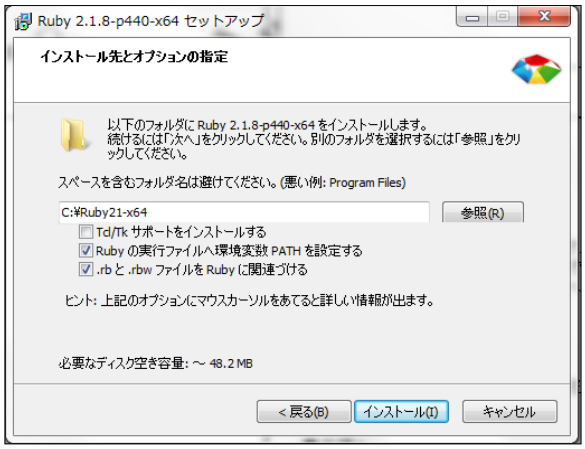
■手順1
https://rubyinstaller.org/downloads/archives/
にアクセスしてください。

■手順2
表示された一覧の真ん中やや下部、
Ruby2.1.8又はRuby2.1.8(x64)を選択します。

■手順.3
適切な箇所にexeを保存します。

■手順.4
ダウンロードしたRuby2.1.8.exeを実行、
セットアップウィザードが実行されるので、
「インストール先とオプションの設定」まで進んでいきます
(日本語 → 同意するにチェック → 次へ)

■手順.5 – コンポーネント選択



■手順.5
インストール先とオプションの設定は
Rubyの実行ファイルへ環境変数PATHを設定する、
.rbと.ebwファイルをRubyに関連付けるにチェックを入れます
インストール先はデフォルトで結構です。
(念のためインストールパスに日本語等は避けたほうが良いです)

■手順.6
完了画面が表示されるので、
完了をクリックしてインストール完了です。

●Rubyの実行方法

Rubyのプログラムを実行するための方法を2つ紹介します。

■ rubyコマンド

テキストエディタを使って作成したプログラムは、rubyコマンドを使って実行します。
コマンドラインからプログラムを作成したフォルダ/ディレクトリに移動し、
「ruby（プログラムファイル名）」と打ち込みプログラムを実行します。
プログラムファイルの拡張子は「.rb」を使用します。

```
MacBook-Pro:~ libertyfish$ cd RubyEducation/
MacBook-Pro:RubyEducation libertyfish$ ruby helloworld.rb
"Hello World!"
MacBook-Pro:RubyEducation libertyfish$
```

■ rubyコマンド　コマンドラインオプション

ruby　[option …][--] [programfile] [argument …]

-c	スクリプト内部形式へのコンパイルのみを行い、実行はしません コンパイル終了後文法エラーがなければ"Syntax OK"と出力します
-d --debug	デバッグモードでスクリプトを実行します
-e script	コマンドラインからスクリプトを指定します -eオプションを付けた時には引数からファイル名を取りません
-h --help	コマンドラインオプションの概要を表示します
--version	Rubyのバージョンを表示します

●Rubyの実行方法

Rubyのプログラムを実行するための方法を2つ紹介します。

■ irb(interactive ruby)

irbはRubyを対話的に入力・実行するためのツールです。メソッドやコードの動作確認が行えます。
簡単な振る舞いの確認ではirbは非常に便利です。

irbを起動するには、コマンドラインより「irb」と入力し、終了するには「exit」もしくは「quit」と入力します。
irbが起動しているときに式を入力し[Enter]を押すと式が即座に実行され返り値を「=>」に続けて表示します

```
irb(main):007:0> puts "hello irb"
hello irb
=> nil
irb(main):008:0> 2 + 3      #式を入力
=> 5
irb(main):009:0> x = 12
=> 12                       #変数の宣言も可能
irb(main):010:0> y = 13
=> 13
irb(main):011:0> x + y   #変数同士の計算
=> 25
irb(main):012:0> def hoge x,y      #メソッドの定義も可能
irb(main):013:1>   p x+y
irb(main):014:1> end
=> nil
irb(main):015:0> hoge(x,y)      #定義したメソッドの呼び出し
25
=> 25
irb(main):016:0> exit           #quit or exitでirbを終了
```

■ irbコマンド　コマンドラインオプション

irb [option] file_name opts

-d	$DEBUGをtrueにします。(ruby -d と同じ)
-r library	ライブラリを読み込みます。
-m	bcモード(分数と行列の計算が出来ます)
--readline	readlineライブラリを使用します。
-h --help	コマンドラインオプションの概要を表示します。
-v --version	irbのバージョンを表示します。
--version	Rubyのバージョンを表示します。

●言語のバージョン管理の必要性

Rubyに限らずプログラミング言語はバージョンの差異によって、
ソフトウェアが正常に動かない場合も多くあります。
開発環境も、複数のソフトウェアやプロジェクトに応じたものをそれぞれ用意する必要があり
言語もそれに準じてバージョンを管理しなくてはなりません。
言語のバージョン管理は、個人的な開発であっても必要不可欠なものとなっているのです。

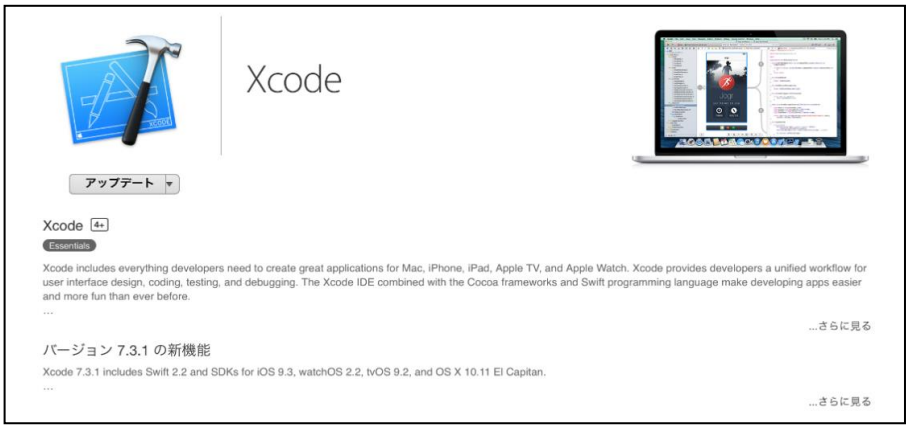
●Rubyのバージョン管理ツール

Rubyでは、RVM・rbenvといったバージョン管理ツールが提供されています。
どちらも、同じバージョン切り替え機能が搭載されていますが、細かな差異があるので
仕様や機能を確認したうえで、使用すると良いでしょう。
本書では、Mac、Linux(Ubuntu)向けのインストール手順を説明します。
Windowsにおけるバージョン管理ツールは存在しますが、少し使いにくいため
Windowsにおけるバージョン管理は、Windowsで開発を行うのではなく
IDE(統合開発環境)を使うことをお勧めします。
Cloud9ならば、RVM・rbenvどちらもLinuxと同じ手順でインストールすることが可能です。

インストール手順は各々の開発環境と使用する管理ツールのページを参照ください。

● [Mac] rbenvインストール手順

1. AppStoreからXcodeをインストールする



2. Homebrewのセットアップを行う

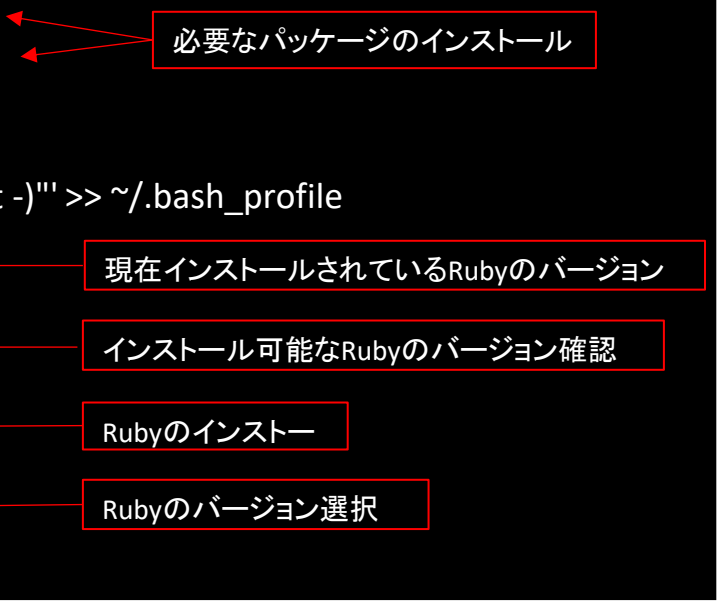
ブラウザでhttp://brew.sh/index_ja.htmlにアクセスし、囲んだ部分をコピーします。
Launchから Terminal.appを起動し、コピーしたものを貼り付け実行します。

```
$ brew doctor
```

もしエラーや警告が出た場合は、ログを参照して対応してください。

3. rbenvのインストール

```
$cd
$pwd
/Users/ユーザー名
$ brew update
$brew install openssl
$brew install readline
$brew install rbenv
$brew install ruby-build
$echo 'eval "$(rbenv init -)"' >> ~/.bash_profile

$rbenv versions

$rbenv install  -l

$rbenv install 2.1.8

$rbenv global 2.1.8
```



● [Ubuntu] rbenvインストール手順

1. インストールに必要なパッケージのインストール

Rubyをインストールするためのパッケージ及びrbenvのインストールに必要な外部パッケージを
インストールします。

```
$ pwd
$ sudo apt-get update
$ sudo apt-get install git
$ sudo apt-get install zlib1g-dev libssl-dev libreadline-dev libyaml-dev libxml2-dev libxslt-dev
$ sudo apt-get install sqlite3 libsqlite3-dev nodejs
```

2. RVMの確認及び削除

rbenvとRVMは共存できません。Ubuntuの初期状態ではRVMはインストールされていませんので、
初めてRubyに挑戦する方やUbuntuをクリーンインストールされた方はこの手順を飛ばしても、
問題ありません。確認はrvmコマンドを実行すれば確認できます。

```
$ rvm
```

上記以外のログが出た場合は下記のコマンドでRVMを削除してください。

```
$ rvm implode
```

3. rbenv及びruby-buildをGitからcloneする

まずは、rbenvをインストールします。

```
$ cd
$ sudo apt-get update
$ git clone git://github.com/sstephenson/rbenv.git .rbenv
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
$ echo 'eval "$(rbenv init -)"' >> ~/.bashrc
$ source ~/.bashrc
```

インストールが完了したら次はruby-buildをインストールします。

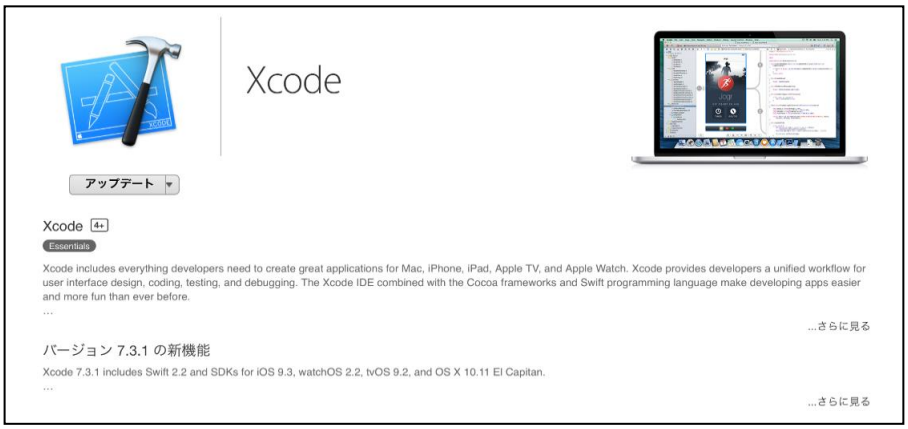
```
$ mkdir -p ~/.rbenv/plugins
$ cd ~/.rbenv/plugins
```

4. rbenvの動作確認及びRubyのインストール

```
$ rbenv -v
$ rbenv
$ rbenv install -l
$ rbenv install 2.1.8
$ rbenv global 2.1.8
$rbenv
```

● [Mac] rvmインストール手順

1. AppStoreからXcodeをインストールする。



2. Homebrewのセットアップを行う

ブラウザでhttp://brew.sh/index_ja.htmlにアクセスし、囲んだ部分をコピーする。
Launchから Terminal.appを起動し,コピーしたものを貼り付け実行する。

```
$ brew doctor
```

もしエラーや警告が出た場合は、ログを参照して対応してください。

3.gpg2をインストール

gpg2がインストールされていないことを確認し、gpg2をインストールします。
既にインストールされている方は飛ばして頂いてかまいません。

```
$ gpg2
gpg2 : command not found
$ brew install gpg2
```

4.Publickeyを登録する。

RVM公式HPの手順通りにまずは公開鍵を作成します(https://rvm.io/rvm/install)

```
$ gpg2 --keyserver hkp://keys.gnupg.net --recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3
```

5.RVMをインストール

公式HPに安定版と開発版のインストールコマンドが記載されているので、
仕様をよく読み、インストールするバージョンを決めましょう。
よく分からなければ、stableで構いません。(https://rvm.io/rvm/install)
curlをインストールしていなければ、curlをインストールしましょう

```
$ curl
curl : command not found
brew install curl
$ curl -sSL https://get.rvm.io | bash -s stable --ruby
```

6. RVMの動作確認

RVMが正常に動作しているかの確認とRubyのインストールを行います。

```
$ rvm  list known
$ rvm install 2.1.8
$ rvm list

$ rvm use 2.1.8
 // デフォルトにしたい場合
$ rvm use 2.1.8  --default
```

インストール可能なバージョン表示

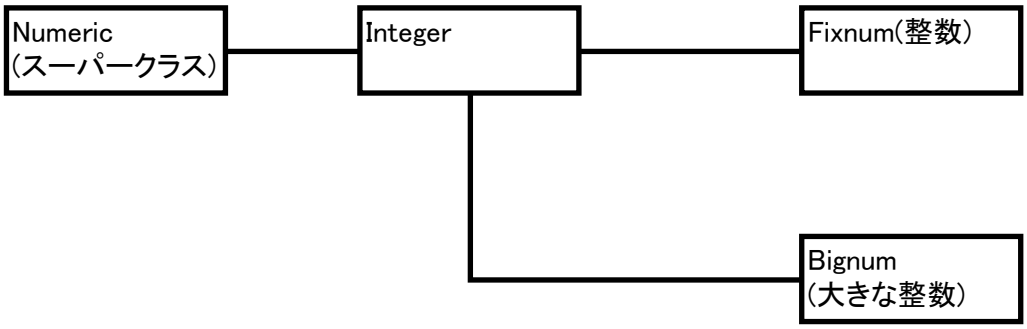
Rubyのインストール

インストール済みのRubyを表示

使用するバージョンの選択

●数値型オブジェクト

数値は「10」や「3.14」など整数や浮動小数点数などのことを言います。



Fixnumクラスが扱えないほど大きな整数の場合、自動的にBignumクラスに変換されます。
なお、Ruby2.4以降で、Fixnum,Bignumは廃止され、Integerクラスに統合されています。
実際の業務でBignumかFixnumか意識されることはほとんどありませんので、
Integer,Fixnum,Bignumは、「数値を扱うクラス」と認識していれば問題ありません。

◆ Mathモジュール

三角関数や対数関数など、よく使う数値演算のためのメソッドは
Mathモジュールで提供されています。
「Math.メソッド名」と、モジュール名を明確に指定するか、
「**include Math**」でインクルードして使います。モジュールについては、13日目の内容を参照してください。

メソッド名	意味
sin(x)	正弦関数(xはラジアン関数)
cos(x)	余弦関数(xはラジアン関数)
tan(x)	正接関数(xはラジアン関数)
asin(x)	逆正弦関数
acos(x)	逆余弦関数
atan(x)	逆正接関数
exp(x)	指数関数
log(x)	自然対数
log10(x)	常用対数
sqrt(x)	平方根

● 配列型オブジェクト

配列は、**複数のオブジェクトを1つにまとめたもの**です。
大量のデータを扱うときや、
複数のデータを次々と自動的に読み出したいときは、配列を使うと便利です。

```
a = [3.14159, "pie", 99]
a.class      #=>  Array
a.length     #=>  3
a[0]         #=>  3.14159
a[1]         #=>  "pie"
a[2]         #=>  99
a[3]         #=>  nil
```

```
b = Array.new
b.class        #=> Array
b.length       #=> 0
b[0] = "second"
b[1] = "array"
b              #=> ["second", "array"]
```

・配列のインデックス(添え字)は0から始まります。
・非負の整数のインデックスで配列を参照すると、その位置のオブジェクトが返されます。
　*その位置に何も無い場合は、nilが返ります。
・負の整数のインデックスで配列を参照すると、配列の終わりから数えた位置にあるオブジェクトが返されます。

```
a = [1, 3, 5, 7, 9]
a[-1]          #=>  9
a[-2]          #=> 7
a[-99]           #=> nil
```

・配列のインデックスは、[start,count]という2つの値の組で特定することも出来ます。
開始位置startから始まるcount個のオブジェクトへの参照が返ります。

```
a = [1, 3, 5, 7, 9]
a[1, 3]          #=>    [3, 5, 7]
a[3, 1]          #=>    [7]
a[-3, 2]         #=>    [5, 7]
```

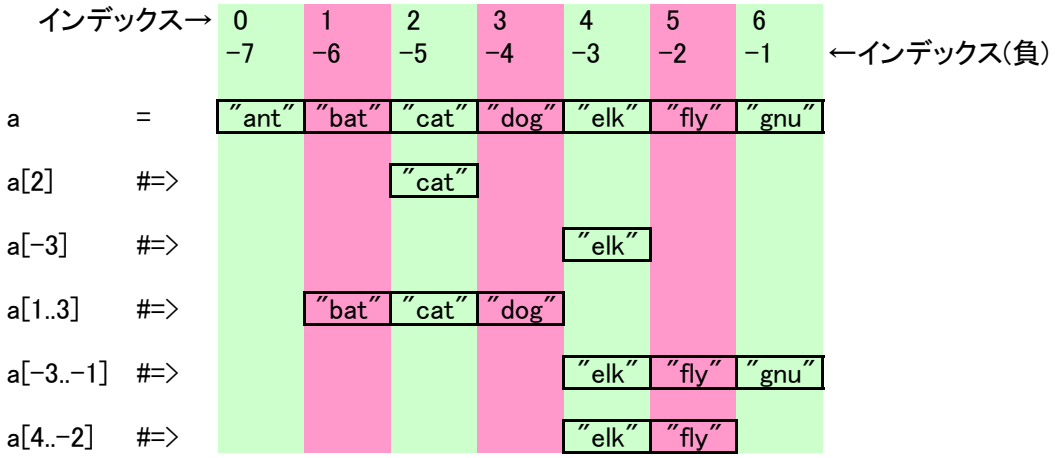
・配列は範囲で参照することも出来ます。範囲は、開始位置と終了位置を
2つまたは3つのピリオドで区切って指定します。
2つのピリオドは終了位置を含みますが、3つのピリオドは終了位置を含みません。

```
a = [1, 3, 5, 7, 9]
a[1..3]          #=>    [3, 5, 7](ピリオド2つなので、終了位置を含む)
a[1...3]         #=>    [3, 5](ピリオド3つなので、終了位置を含まない)
a[3..3]          #=>    [7]
a[-3..-1]        #=>    [5, 7, 9]
```

・[ ]演算子には対応する[ ]= 演算子があり、これを使って配列の要素に代入できます。
[ ]= のインデックスに整数を1つだけ指定した場合は、その位置の要素が代入文の右辺の値で変換されます。代入によって配列に隙間が出来た場合は、nilで埋められます。

```
a  = [1, 3, 5, 7, 9]      #=>    [1, 3, 5, 7, 9]
a[1] = 'bat'              #=>    [1, "bat", 5, 7, 9]
a[-3] = 'cat'             #=>    [1, "bat", "cat", 7, 9]
a[3] = [9, 8]             #=>    [1, "bat", "cat", [9, 8], 9](多次元配列)
a[6] = 99                 #=>    [1, "bat", "cat", [9, 8], 9, nil, 99]
                                   →(添え字5の部分に隙間が出来たため)
```

イメージは、こんな感じです。



● ハッシュ型オブジェクト

・ ハッシュとは、インデックス付けられたオブジェクトの集まりです。
配列と良く似ていますが、配列が整数でしかインデックス付けできないのに対し、
ハッシュは任意の型のオブジェクト(文字列、正規表現等)でインデックス付けできます。

◆ ハッシュを定義するには、次のようにします。

```
オブジェクト = {キー1: 値1, キー2: 値2, …}
```

◆ キーに関連付けられた値を取得するには、次のようにします。

```
h = {apple:  150, banana: 300, lemon: 300}

p h[:apple]
p h[:banana]
p h[:lemon]
p h[:orange]
```

出力

```
150
300
300
nil
```

・ 格納されていないキーを参照すると、nil が返ります。

◆ ハッシュに要素を追加するには、次のようにします。

・ new後に追加

```
h = Hash.new

h['apple'] = 150
h['banana'] = 200
h['lemon'] = 300

p h['apple']
p h['banana']
p h['lemon']
```

出力

```
150
200
300
```

・ { }による定義の後に追加

```
h = {apple: 150, banana: 300, lemon: 300}

h['orange'] = 500
```

出力

```
500
```

・ **store**メソッドを使用する

ハッシュオブジェクト.store(キー, 値)という使い方で、オブジェクトに要素を保存できます。

```
h = Hash.new

h.store('apple',150)
h.store('banana', 200)
h.store('lemon', 300)

p h['apple']
p h['banana']
```

出力

```
150
200
300
```

◆ length または size メソッドで、ハッシュの要素数を取得できます。

```
h = {apple: 150, banana: 300, lemon: 300}

p h.length
p h.size
```

出力

```
3
3
```

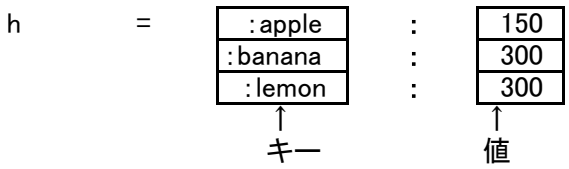
◆ ハッシュに対してブロックを実行する

・ キーを引数としてブロックを実行するには**each_key**メソッド、値を引数として実行するには**each_value**メソッド、キーと値を引数として実行するには**each**メソッドまたは**each_pair**メソッドを使用します。

```
h = {apple: 150, banana: 300, lemon: 300}

h.each_key {|key| puts key}
h.each_value {|value| puts value}
h.each_pair {|key, value| puts "#{key}: ¥¥#{value}"}
```

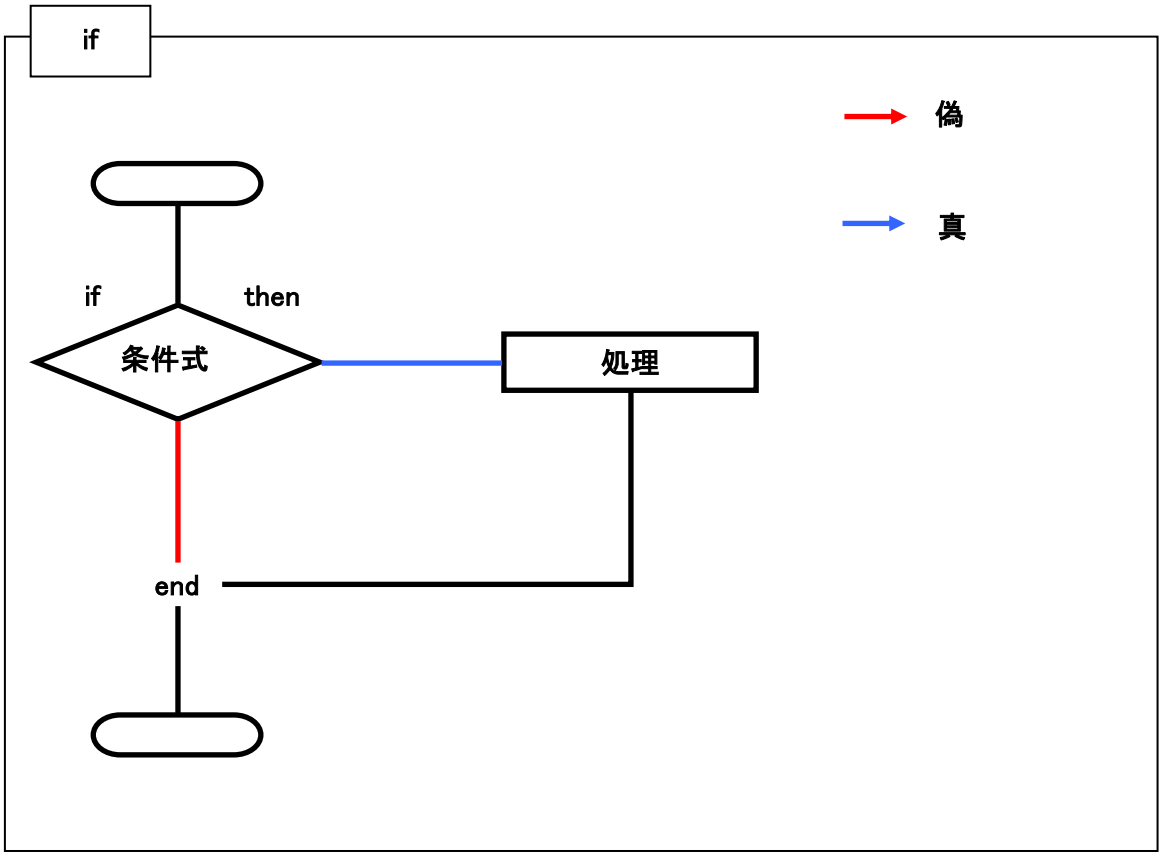
出力

```
apple
banana
lemon
150
300
300
apple: ¥150
banana: ¥300
```



● **制御文**
◆ 基本的な制御文に、次のようなものがあります。

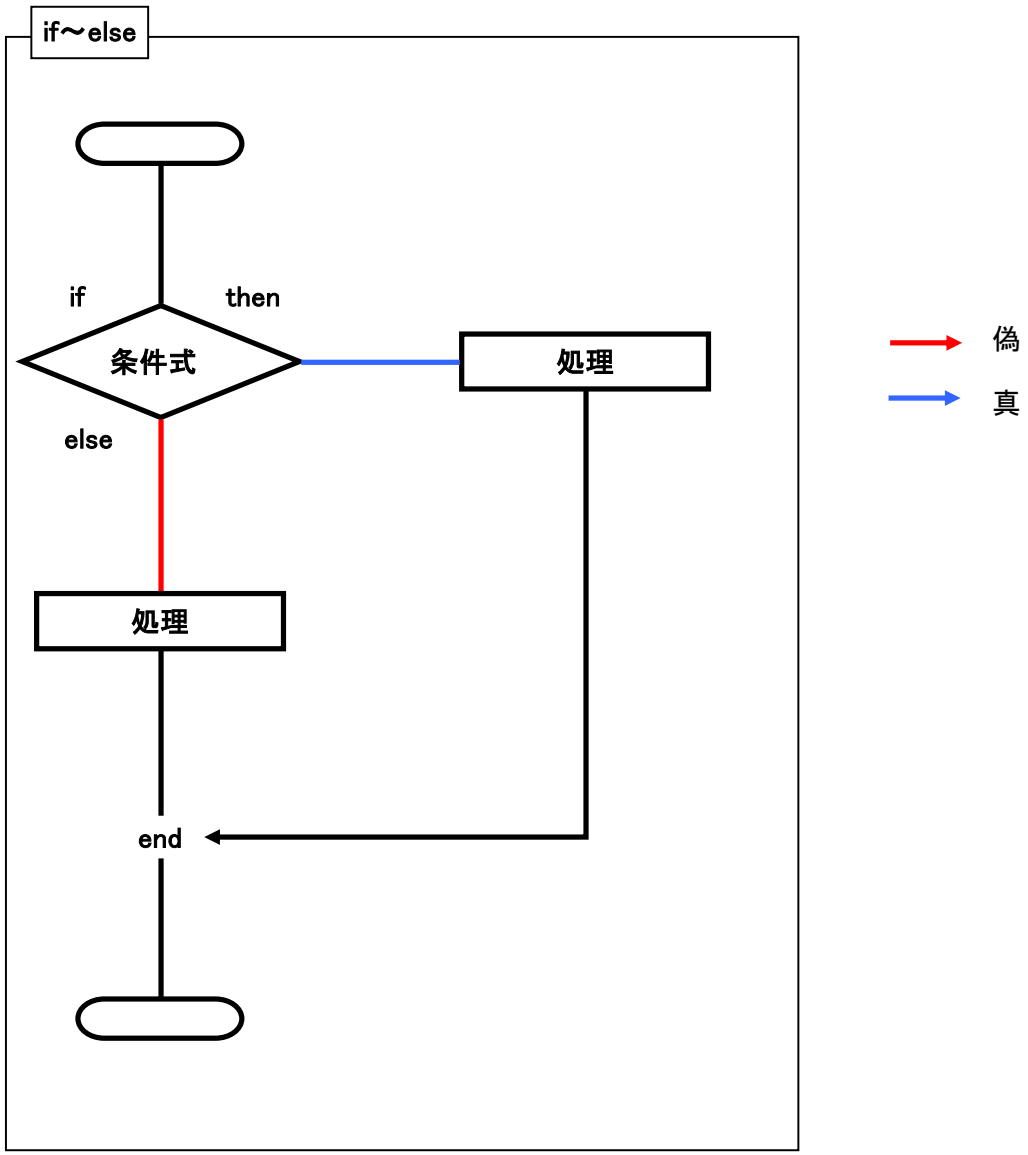
● **if文**




```
if 条件式 then

  条件式が成り立つときに実行する処理

end
```

条件式が真のときに処理を行います。
条件式の後に改行がある場合は、「then」を省略することが可能です。

● if文～else文

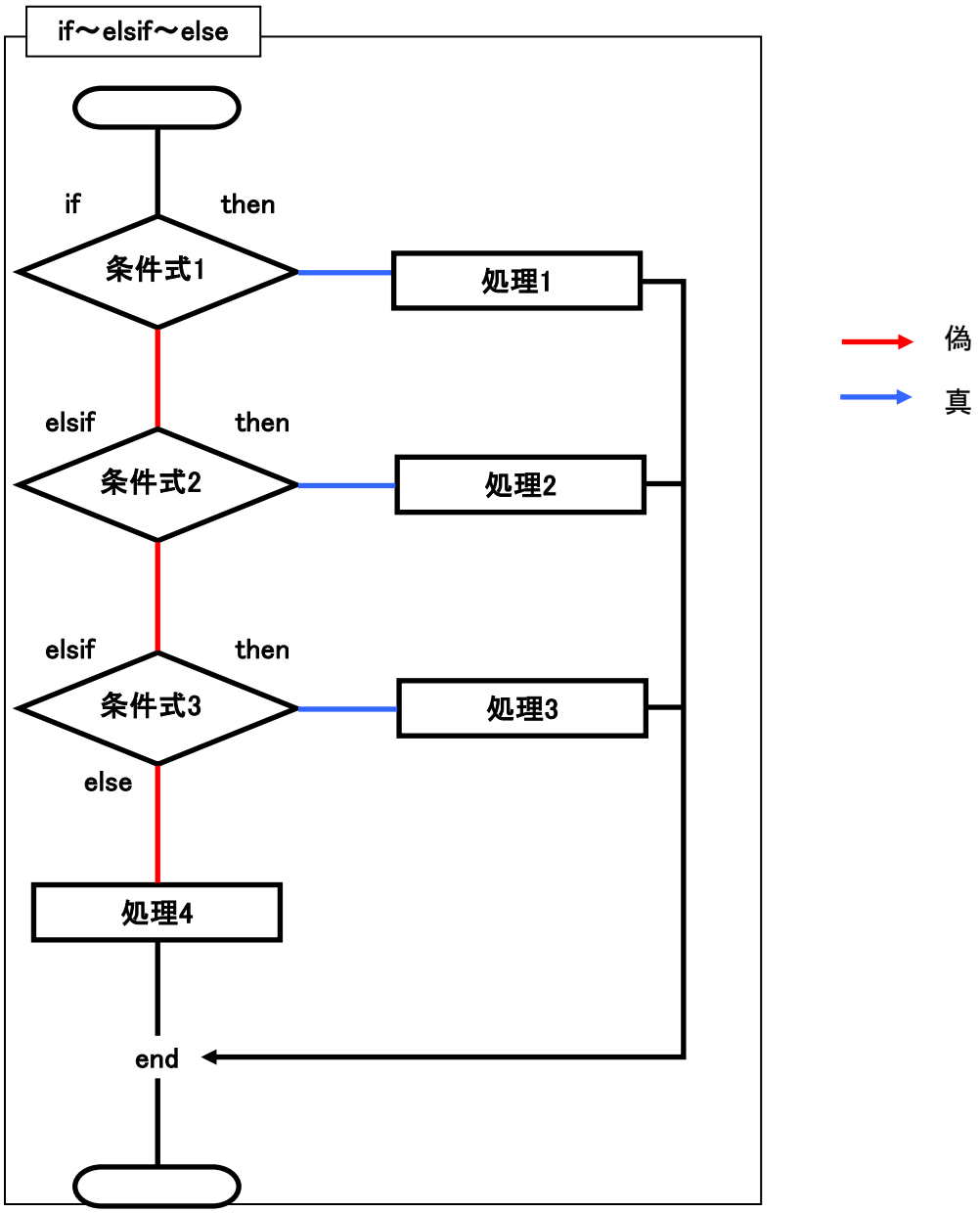


条件式を評価し、真(true)だった場合には「then」から「else」までに書かれた処理を上から順に実行します
そして条件式が偽(false)だった場合には「else」から「end」までに書かれた処理を上から順に実行します。

```
if 条件式 then
  条件式が成り立つときに実行する処理
else
  条件が成り立たないときに実行する処理
end
```

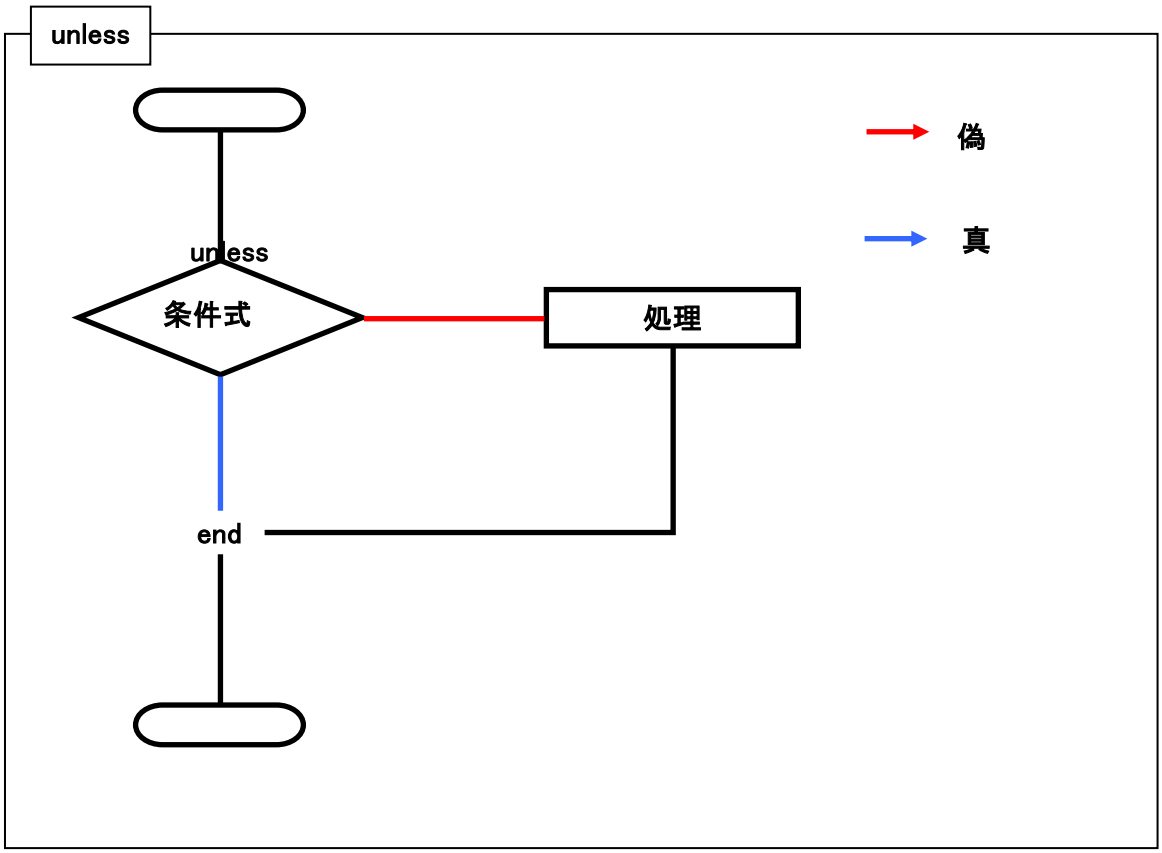
● if文～elsif～else文

複数の条件分岐をさせたいときに使用します。




```
if 条件式1 then
  条件式1が真の時に実行する処理(処理1)
elsif 条件式2 then
  条件式1が偽で条件式2が真の時に実行する処理(処理2)
elsif 条件式3 then
  条件式1及び条件式2が偽で条件式3が真の時に実行する処理(処理3)
else
  全ての条件式が偽の時に実行する処理(処理4)
end
```

● unless文

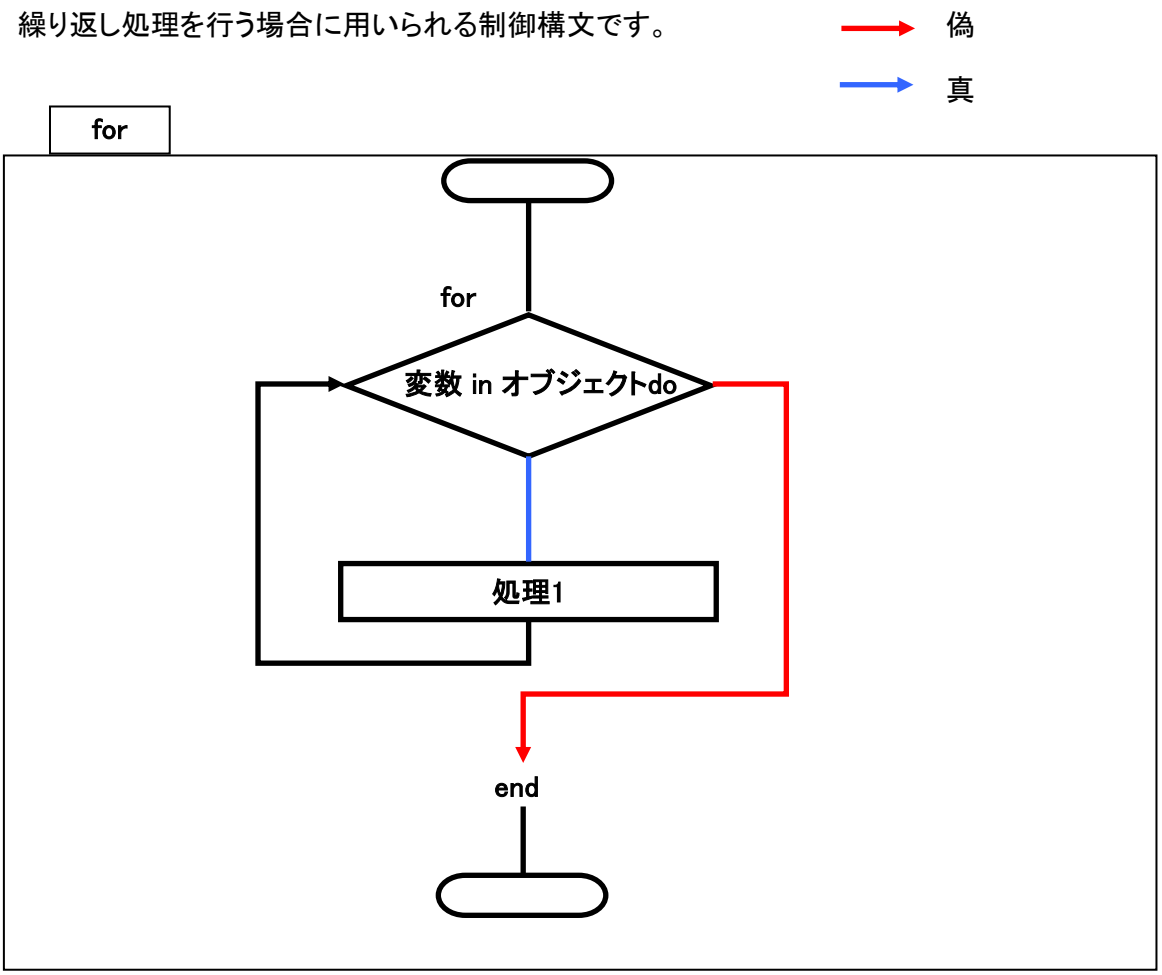



```
unless 条件式 then

  条件式が成り立たないときに実行する処理

end
```

条件式が偽のときに処理を行います。つまり、条件が成り立つときは何もしません。
条件式の後に改行がある場合は、「then」を省略することが可能です。

● for文

繰り返し処理を行う場合に用いられる制御構文です。



上記の記法は、配列などから、要素を一つずつ取り出して処理を行いたいときに使われます。

```
for 変数 in 配列や範囲
  実行したい処理
end
```

```
a = [1,2,3]

for i in a
   print i, "¥n"
end
```

出力

```
1
2
```

配列のオブジェクトaの要素数だけ、繰り返します。
この例では3回処理が繰り返されます。

● 演算子には、以下のようなものがあります。

演算子一覧表　(優先度順)

[ ] , [ ]=	配列要素の参照・設定
**	べき乗
! , ~ , + , −	否定・補数・単項プラス・単項マイナス
* , / , %	乗算・除算・剰余
+ , −	加算・減算
>> , <<	右シフト・左シフト
&	ビット演算の論理積
^ , |	ビット演算の排他的論理和・通常の論理和
<= , >= , > , <	比較演算子
<=> , == , === , != , =~ , !~	等号、パターンマッチ演算子
&&	論理積
||	論理輪
.. , ...	境界を含む範囲・境界を含まない範囲
a ? b : c	三項演算子 if a then b else c
= , %= , ^= , /= , −= , += , |= , &= , >>= , <<= , *= , &&= , ||= , **=	代入演算子
defined?	シンボル定義のチェック
not	論理否定
or , and	論理和・論理積
if , unless , while , until	式修飾子
begin / end	begin式

＊　インクリメント･デクリメント演算子はありません。+= と −= を使用のこと

● ユーザー入出力

◆ 標準入出力

・ 入出力の対象になるデータは文字列、つまりStringオブジェクトです。
入力操作を行うと、データの先頭から終わりまでが順に読み込まれ、
出力を行うと、書き込んだ順に次々と追加されていきます。

入力用メソッド一覧

```
io.gets(rs)
io.each(rs)
io.readlines(rs)
io.lineno
io.lineno=(number)
io.getc
io.each_byte
io.ungetc(ch)
```

出力用メソッド一覧

```
io.puts(str0,str1,…)
io.putc(ch)
io.print(str0,str1…)
io.printf(fmt,arg0,arg1,…)
```

◆ コマンドライン入力

・ コマンドラインの情報をデータとして受け取るには、ARGVという配列を使用します。
このARGVという配列は、コマンドラインからスクリプトの引数として与えられた
文字列を要素として持っています。

なお、受け取るデータは文字列ですので、数値として扱いたい場合はto_iメソッドを使用します。

◆ 文字化け

Shift-JISへの対応は、次のようにします。

1） -Ksオプションをつけて実行する

```
ruby -Ks foo.rb
```

2） プログラムの先頭に、以下の行を追加する

```
#！ Ruby -Ks
```

3） 変数$KCODEを設定する

```
$KCODE = "SJIS"
```

● Timeクラス

◆ 時刻を扱うクラスです。現在の時刻を取得したり任意の時刻を
設定したりする事が出来ます。

Time.newメソッドまたはTime.nowメソッドを使って、現在の時刻を表す
Timeオブジェクトを得ることが出来ます。

```
t1 = Time.new
p t1

t2 = Time.now
```

出力

```
Fri Jan 21 15:32:00 +0900 2021
Fri Jan 21 15:32:00 +0900 2021
```

+0900は、世界標準時との時差です。

どちらも同じですが、「new」メソッドはTimeクラスのオブジェクトを作成して
初期値として現在時刻が設定されているというのに対して、
「now」メソッドの場合は現在時刻を表すTimeクラスのオブジェクトを作成します。
その為、現在時刻を取得したいという場合であれば、
「now」メソッドを使った方が意味合いとしては正しいです。

◆ Timeオブジェクト同士は、比較したり差を求めたりすることが出来ます。

```
t1 = Time.now
sleep(10)
t2 = Time.now
p t1 < t2
p t2 - t1
```

出力

```
true
10.0
```

生成された時間差を求めています。

◆ Timeクラスには、次のようなメソッドが用意されています。

メソッド	取得できる時刻要素
sec	秒を整数で取得
min	分を整数で取得
hour	時を整数で取得
mday	日を整数で取得
day	mdayの別名
mon	月を整数で取得
month	monの別名
year	年を整数で取得
wday	曜日を0(日曜日)から6(土曜日)の整数で取得
yday	1月1日からの通算日を整数で取得
isdst	夏時間があるなら true
dst?	isdstの別名
zone	タイムゾーンを表す文字列で取得

これらのメソッドを使用すれば、年や月などの時刻を構成する各要素を取り出すことが出来ます。

```
youbi = ['日', '月', '火', '水', '木', '金', '土']

t = Time.now

print(t.year, "年", t.month, "月", t.day, "日¥n")
print(youbi[t.wday], "曜日¥n")
print(t.hour, "時", t.min, "分", t.sec, "秒¥n")
print("TimeZone:", t.zone, "¥n")
print("今年の元旦から数えて", t.yday, "日目¥n")
```

出力

```
2021年3月23日
火曜日
9時40分6秒
TimeZone:東京 (標準時)
今年の元旦から数えて82日目
```

◆ 時刻をフォーマットに従った文字列にしたいときは、Timeクラスの**strftime**メソッドを使います。

フォーマット	意味と範囲
%A	曜日の名称(Sunday,Monday,...)
%a	曜日の省略名(Sun,Mon...)
%B	月の名称(January,February,...)
%b	月の省略名(Jan,Feb,...)
%c	日付と時刻
%d	日(01～31)
%H	24時間制の時(00～23)
%I	12時間制の時(01～12)
%j	年中の通算日(001～366)
%M	分(00～59)
%m	月を表す数字(01～12)
%p	午前または午後(AM, PM)
%S	秒(00～60) *60は閏秒の場合
%U	週を表す数。最初の日曜日が第一週の始まり(00～53)
%W	週を表す数。最初の月曜日が第一週の始まり(00～53)
%w	曜日を表す数。日曜日が0(0～6)
%X	時刻
%x	日付
%Y	西暦を表す数
%y	西暦の下2桁(00～99)
%Z	タイムゾーン(JST)など
%z	タイムゾーン(+0900)など
%%	%をそのまま出力

```
t = Time.now
p t.strftime("%Y/%m/%d  %H:%M:%S")
p t.strftime("%a %b %d  %H:%M:%S %Z %Y")

p t.to_s
```

出力

```
"2021/01/22 15:34:47
"Sat Jan 22  15:34:47 東京（標準時） 2021"
"Sat Jan 22  15:34:47 +0900 2021"
```

◆ Dateクラス

・ 日付を扱うクラスです。時刻は扱いません。
Dateクラスは組み込みクラスとなっていないため、このクラスを
使用するにはrequireメソッドを使用して、ライブラリを読み込む必要があります。

```
require "date"

d = Date.new(2018, 2 ,4)
print(d, "¥n")

dt = Date.today
```

出力

```
2018-02-04
2021-01-29
```

上は指定した日付を引数として与えています。下は生成された日の日付を返しています。

・Dateクラスでnewメソッドを使う場合、年月日の各値を指定しますが、
日引数に負の値を指定した場合は
「その月の最後の日から何日前の日付なのか」を指定したことになります。

```
require "date"

day1 = Date.new(2020, 6 ,-1)
print(day1, "¥n")
day2 = Date.new(2020, 7 ,-1)
print(day2, "¥n")
day3 = Date.new(2020, 8 ,-1)
print(day3, "¥n")
```

出力

```
2020-06-30
2020-07-31
2020-08-31
```

・ 二つのDateオブジェクトの差分は日を単位としていますので、それらを引き算すると、その間の日数が得られます。また、Dateオブジェクトに整数を足したり引いたりするとその前後の日付を得ることが出来ます。

```
require "date"
d = Date.today

puts d
puts d + 1
puts d + 100
puts d - 1
```

出力

```
2021-03-23
2021-03-24
2021-07-01
2021-03-22
```

・ また、Dateクラスのオブジェクトを作成すると、そのオブジェクトから
年や月など個々の日付要素を取り出すことができます。
そのためのメソッドが、下記のようなものです。

メソッド	取得できる日付要素
mday	日を整数で取得(1-31)
day	mdayの別名
mon	月を整数で取得(1-12)
month	monの別名
year	年を整数で取得
wday	曜日を0(日曜日)から6(土曜日)の整数で取得
yday	1月1日からの通算日を整数で取得(1-366)
cweek	暦週を整数で取得(1-53)

```
require "date"

youbi = %w[日 月 火 水 木 金 土]

d = Date.today

print(d.year, "年", d.month, "月", d.day, "日¥n")
print(youbi[d.wday], "曜日¥n")
print("本日は今年の", d.cweek, "週目¥n")
print("1月1日から数えて", d.yday, "日目¥n")
```

出力

```
2021年3月23日
火曜日
本日は今年の12週目
1月1日から数えて82日目
```

◆ **DateTimeクラス**

- 日付と時刻を扱います。Timeクラスと同様のことが行えますが、内部仕様が異なります。
  Timeクラスはシステムの環境の制限を受けますが、DateTimeクラスはシステムの制限を受けない
  より一般的な時刻を扱うことが出来ます。Dateクラスも同様です。
  このクラスも、組み込みメソッドではないため、使用の際には**require**でライブラリを
  呼び出す必要があります。

```
require "date"

d = DateTime.new(2021, 2 ,4, 16, 20, 45, 0.375)
print(d, "¥n")

tn = DateTime.now
print(tn)
```

出力

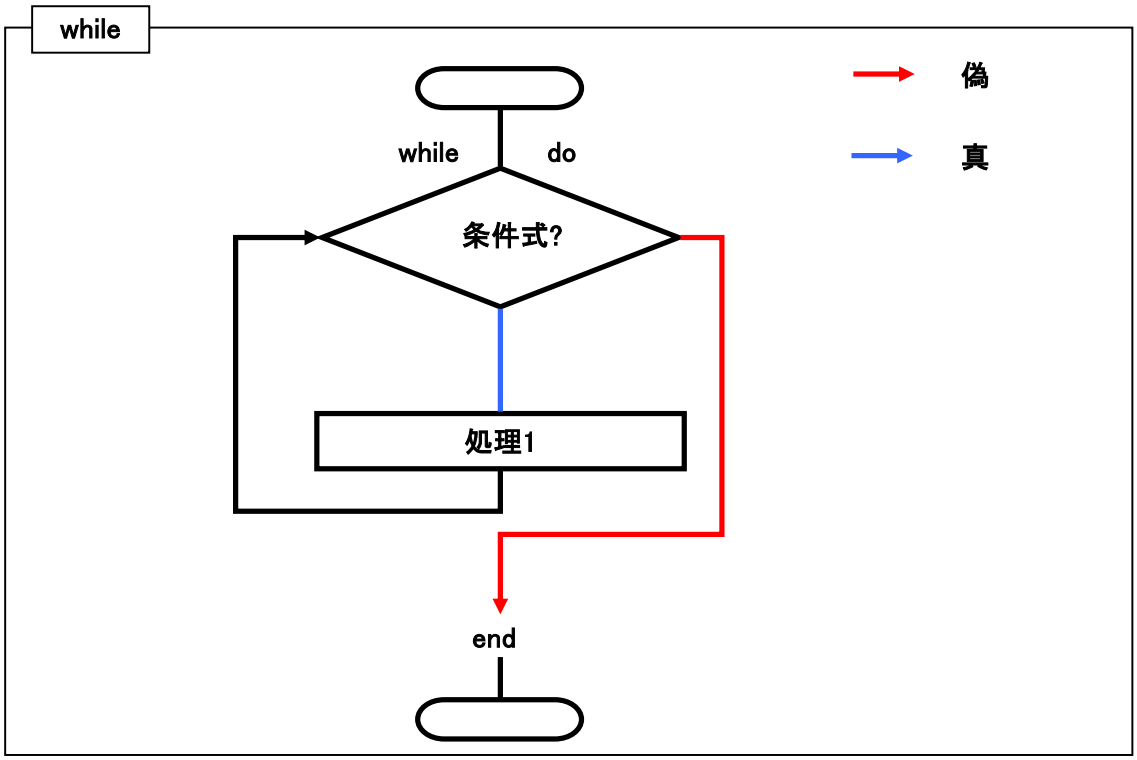
```
2021-02-04T16:20:45+09:00
2021-03-23T09:39:04+09:00
```

newの引数は順に年、月、日、時、分、秒、オフセット(時差)となっています。
0.375は、「時差/24」の結果です。東京時間なら世界標準時とは9時間差ですから,
9/24 = 0.375です。

● while文

条件式が真の間繰り返し処理を行う構文です。

- 条件式が真（true）となる間は「do」から「end」までに記述された文が上から順に実行されループ処理部に記述された処理が一通り実行され終わると、改めて条件式の判断が行われます。これを条件式が偽(false)になるまで繰り返します。
  ※「do」は省略可能です。




```
while 条件式 do

実行する処理

end
```

```
i = 1
while i < 3
 print i, "¥n"
 i += 1
```

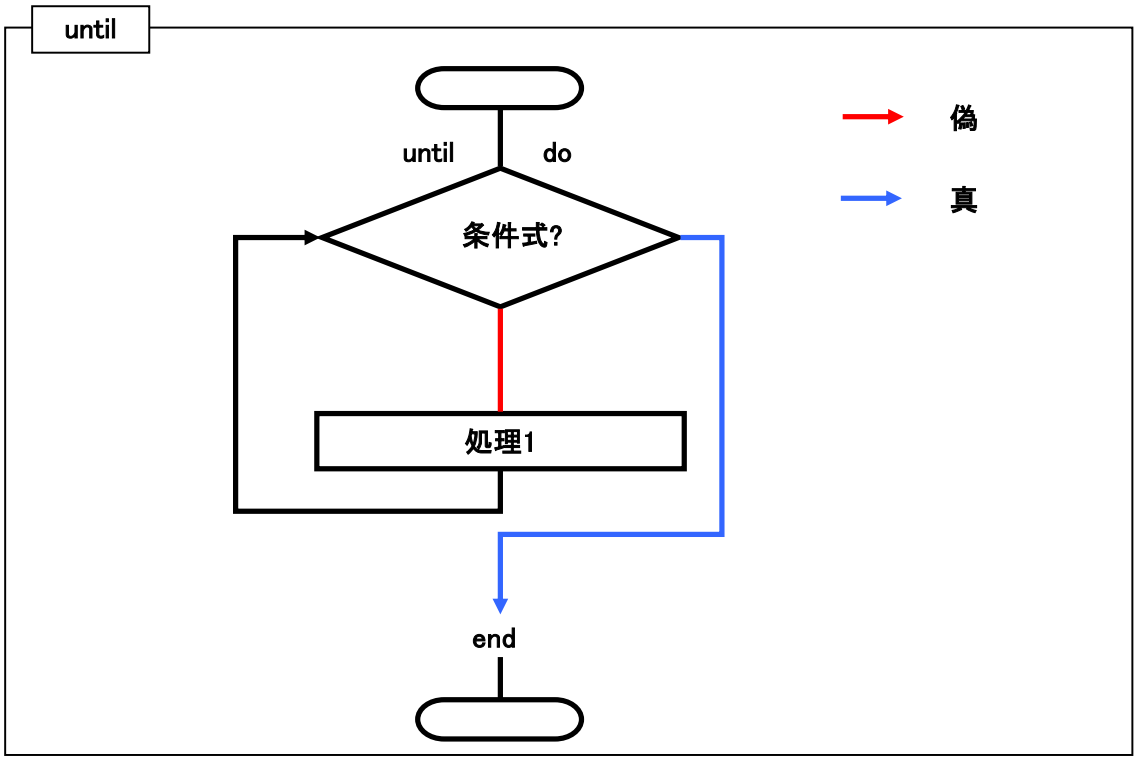
出力

```
1
2
```

● until文

条件式が偽の間繰り返し処理を行う構文です。

- 条件式が偽(false)となる間は「do」から「end」までに記述された文が上から順に実行されます。
  ループ処理部に記述された処理が一通り実行され終わると、改めて条件式の判断が行われます。
  これを条件式が真(true)になるまで繰り返します。
  ※「do」は省略可能です。



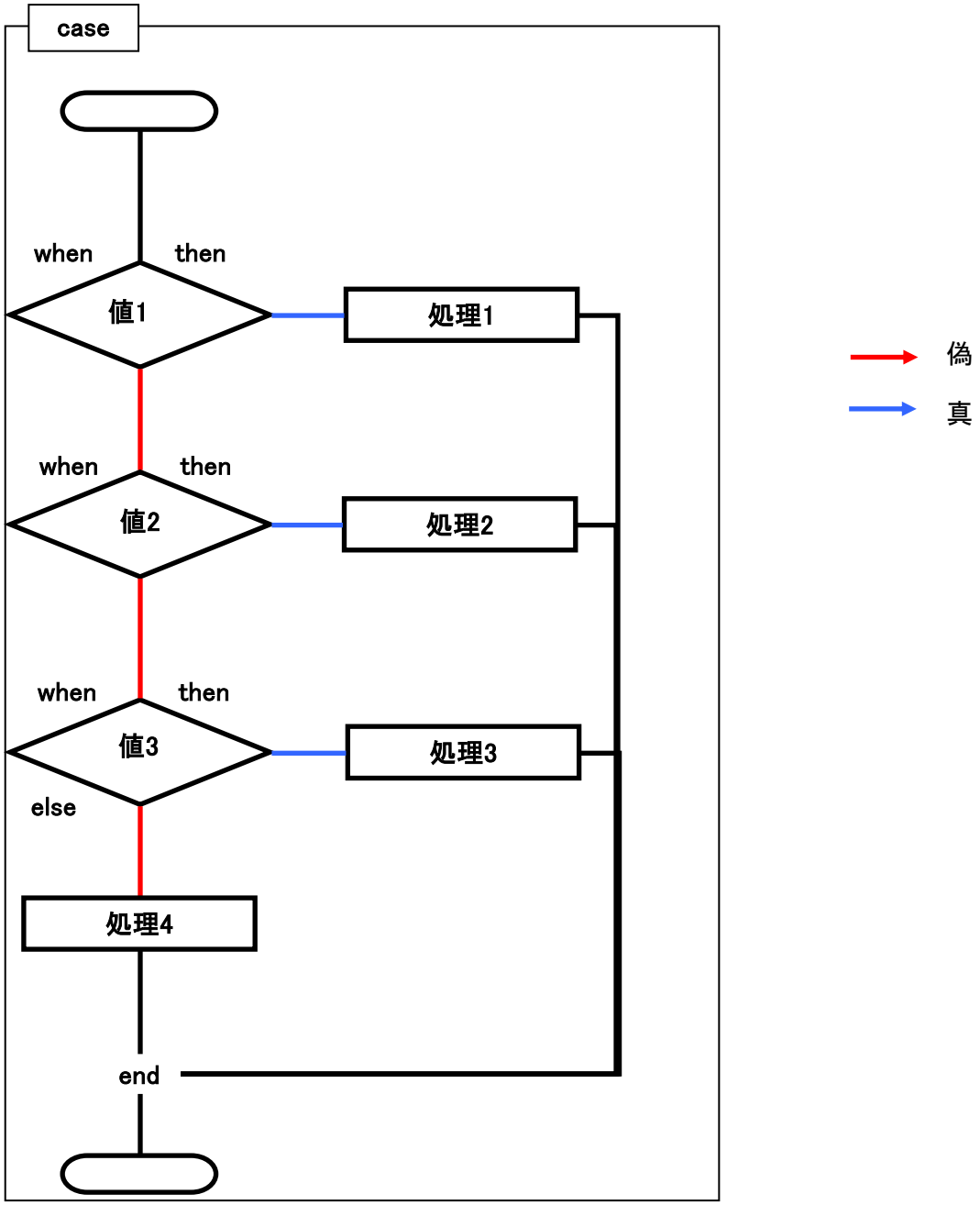

```
until 条件式 do
 条件が偽の場合実行する処理
 end
```

```
sum = 0
i = 1
until sum >= 50
 sum += i
 i += 1
end
```

出力

```
55
```

● case文　　複数の条件分岐を行いたい時などに使用します。




```
case 対象オブジェクト
when 値1 then
  値1と一致する場合に行う処理(処理1)
when 値2 then
  値2と一致する場合に行う処理(処理2)
when 値3 then
  値3と一致する場合に行う処理(処理3)
else
  どの値にも一致しない場合に行う処理(処理4)
end
```

● 修飾子

「ｉｆ」文を使うことで条件に応じて行う処理を実行することが出来ますが、
次のような記述方法も可能です。

```
真の時に実行する式 if 条件式
```

このような構文を「if」修飾子といいます。
上記では条件式が真の場合に「if」の左辺の式を実行します。

これは「ｉｆ」文で次のように記述した場合と同じです。

```
if 条件式 then
  真の時に実行する式
end
```

「if」修飾子は実行する処理を先に記述しているだけです。記述例は、こんな感じ(例1)

例1)

```
print("num = ", num) if debug
```

変数「debug」が真の場合だけ変数「num」の値を出力しています。

なお、次のように「if」文を使って同じ事が記述できますが、
より簡潔に記述でき何をしようとしているのかも明確に出来ます(例2)

例2)

```
if debug then
  print("num = ", num)
end
```

例1を、条件式を前置で記述したものが例2です。
こちらの書き方の方が、わかり易いかも知れません。

なお「if」修飾子と同じように「unless」修飾子も用意されています。

```
偽の時に実行する式 unless 条件式
```

「unless」修飾子は条件式が偽の場合に左辺の式を実行します。

「while」文を使用することで、条件が真の間繰り返し処理を実行させることが出来ますが、
次のような記述方法も可能です。

```
真の時に実行する式 while 条件式
```

このような構文を「while」修飾子といいます。
上記では条件式が真の間「while」の左辺の式を繰り返し実行します。

これは「while」文で次のように記述した場合と同じです。

```
while 条件式 do
  真の時に実行する式
end
```

「while」修飾子は実行する処理を先に記述しているだけです。

例1）

```
num = 2
num *= 2 while num < 1000
```

2の累乗を計算していき1000を超えたらストップします。

条件式を後置するメリットは、
下記のように条件式を前置する場合と比較してよりコンパクトに記述できるという点です。

例2）

```
num = 2
while num < 1000 do
  num *= 2
end
```

例1を、条件式を前置で記述したものが上記「例2」です。少しかさばります。

なお「while」修飾子と同じように「until」修飾子も用意されています。

```
偽の時に実行する式 until 条件式
```

「until」修飾子は条件式が偽の間「until」の左辺の式を繰り返し実行します。

● 繰り返しの制御

◆ 繰り返し処理を行う構文には、break,next,redo,retryなどといった繰り返しを制御する命令があります。

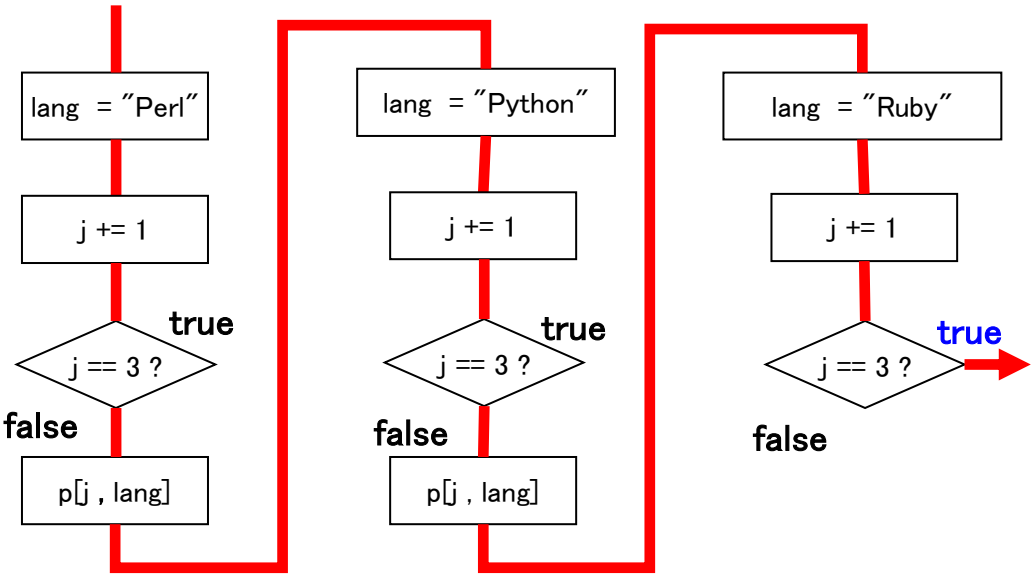
● break

・ 繰り返しを中断し、ループの中から抜け出します。

下記例では、j が3のときに break 命令を実行し、for ループから抜け出します。

```
looplang = ["Perl", "Python", "Ruby", "Scheme"]
j = 0
for lang in looplang
  j += 1
  if j == 3
    break
  end
  p [j, lang]
```

＊ 例文に記述されている「p」は、埋め込みメソッドと呼ばれるもののーつです。
「p」は、オブジェクトの内容を適切な形で表示してくれるという便利なメソッドです。



j が3まで加算されると、break命令が実行され、ループから抜け出します。
そのため、「Ruby」「Scheme」の文字列は出力されません。

出力

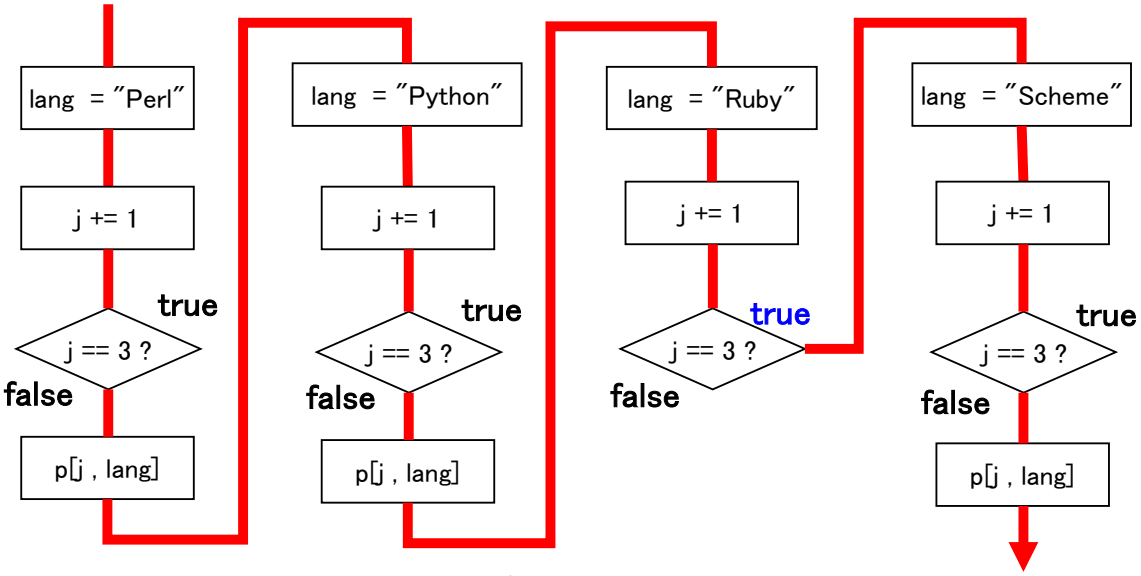
```
[1, "Perl"]
[2, "Python"]
```

● next

◆ next以降の処理を飛ばして、次の回の処理を開始します。

下記例では、j が 3 の時にnext が実行され、for 文の次の繰り返しに移ります。

```
looplang = ["Perl", "Python", "Ruby", "Scheme"]

j = 0
for lang in looplang
  j += 1
  if j == 3
    next
  end
  p [j, lang]
end
```



j が 3 の時にnext が実行され、for ループの次の繰り返しに移ります。
つまり、j=3 の時の p[j, lang] 処理は実行せず、次のループ処理へ移ります。
そのため、「Ruby」は表示されず「Scheme」が表示されることになります。

出力

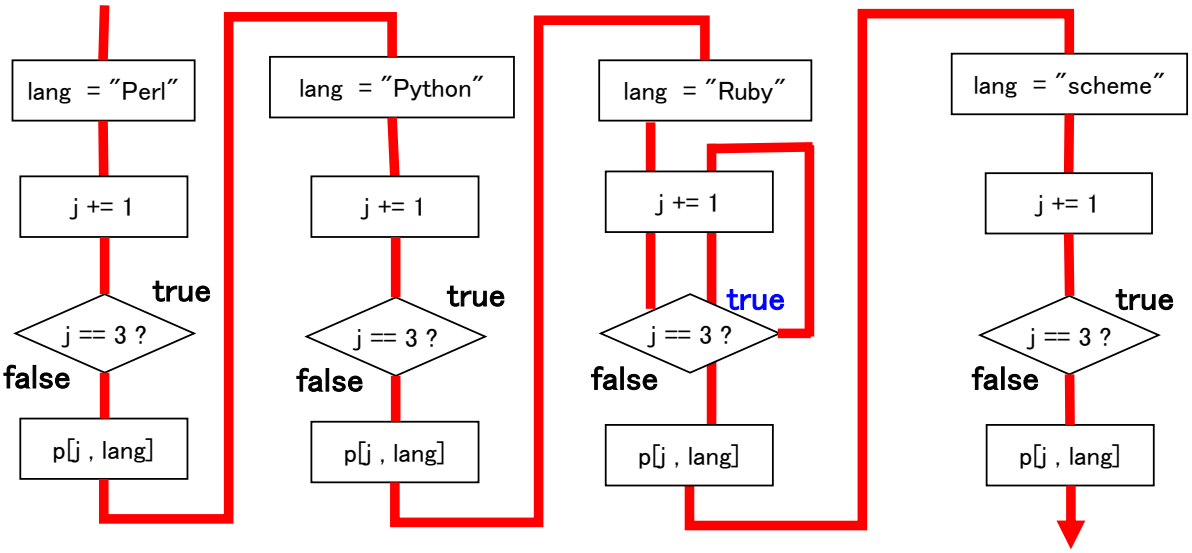
```
[1, "Perl"]
[2, "Python"]
[4, "Scheme"]
```

● **redo**

◆ redoが実行された繰り返しの回の処理を、もう一度繰り返します。

下記例では、j が 3 の時に redo が実行され、その回の処理がもう一度繰り返されます。

```
looplang = ["Perl", "Phyton", "Ruby", "Scheme"]

j = 0
for lang in looplang
  j += 1
  if j == 3
    next
  end
  p [j, lang]
end
```



j が 3 の時にredo が実行され、その回のeachメソッドがもう一度実行されます。
つまり、「j += 1」がもう一度実行されて j が4になり、「j == 3」が成り立たなくなるので、
「 p [j, lang] 」も無事に実行されることになり、「Ruby」の文字列の表示も行われます。

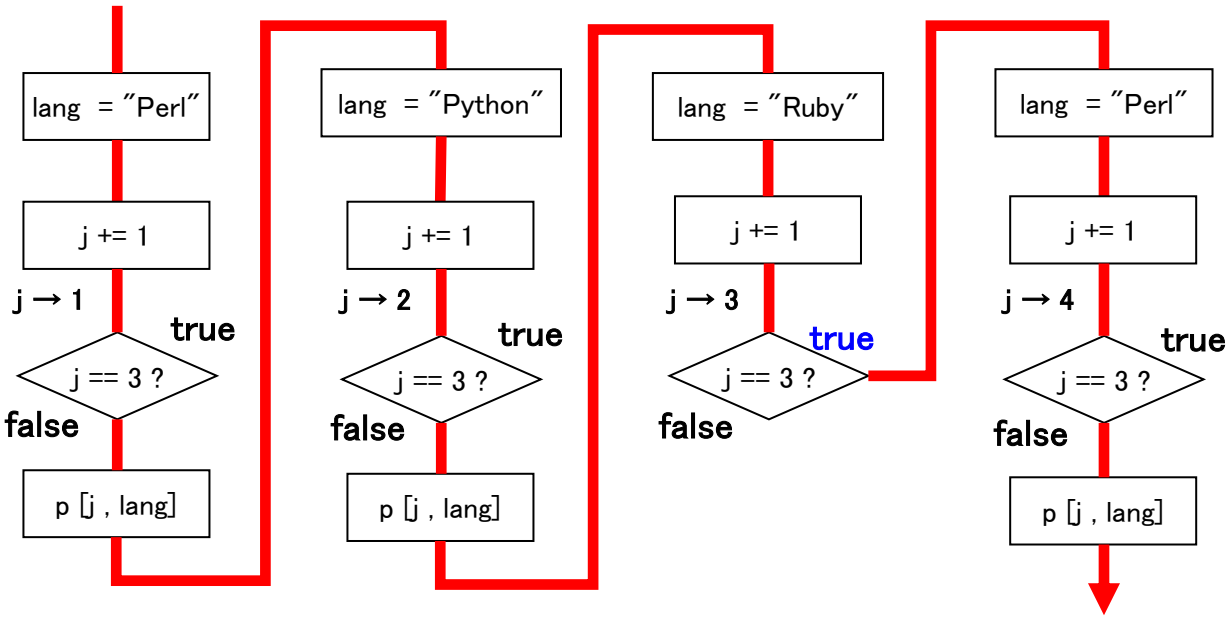
出力

```
[1, "Perl"]
[2, "Phyton"]
[4, "Ruby"]
[5, "Scheme"]
```

● retry

・ その名のとおり、ループを最初からやり直します。
次の例では、j が 3 になると retry でループの最初に戻り、再び配列の先頭 "Perl" から
ループが行われます。

```
looplang = ["Perl", "Phyton", "Ruby", "Scheme"]
j = 0
for lang in looplang
  j += 1
  if j == 3
    retry
  end
  p [j, lang]
end
```



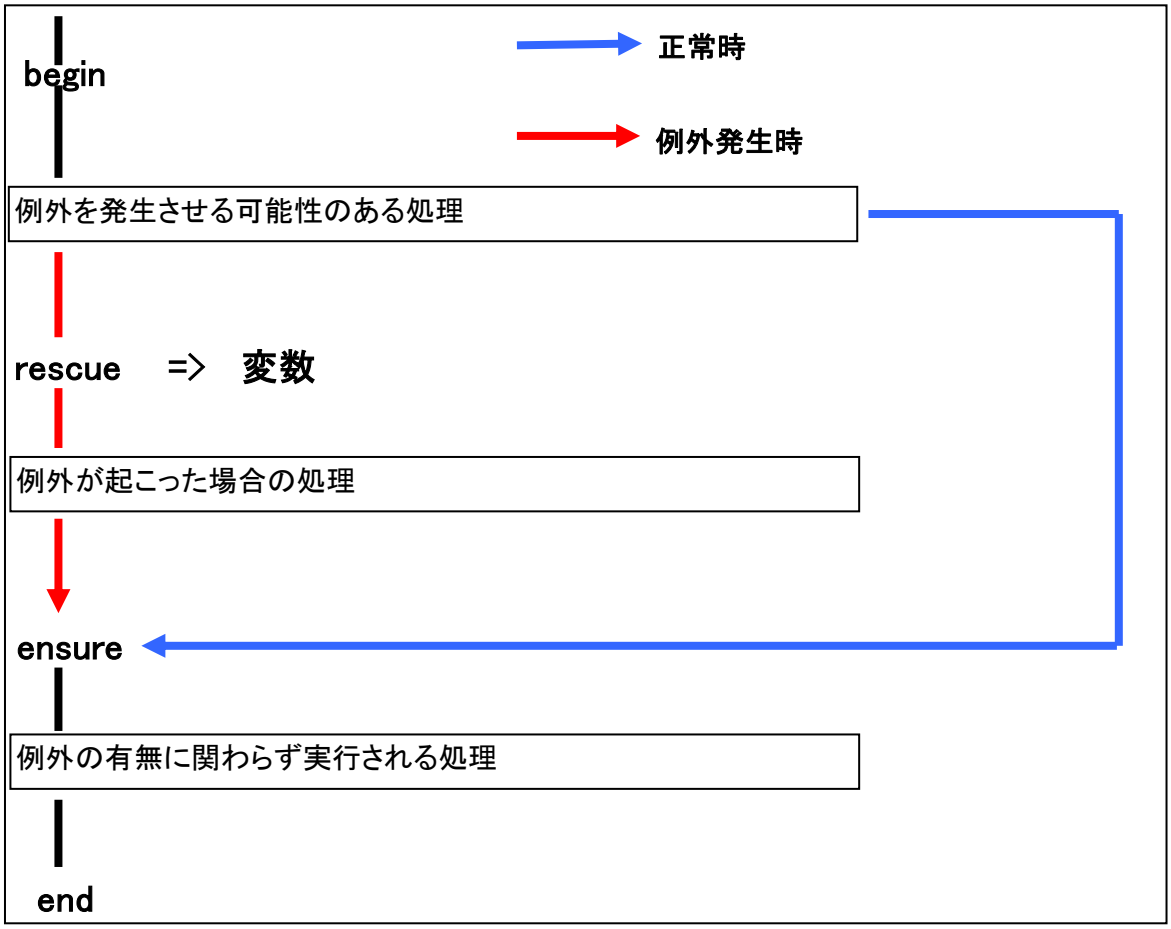
・ 無限ループにならないように注意してください。

出力

```
[1, "Perl"]
[2, "Phyton"]
[4, "Perl"]
[5, "Phyton"]
[6, "Ruby"]
[7, "Scheme"]
```

● 例外処理

◆ 例外処理を行います。



- Rubyでは、例外もオブジェクトとして扱われます。rescueに続けて変数名を指定することで、発生した例外を例外オブジェクトとして得ることができます。

- 例外オブジェクトには、次のようなメソッドが用意されています。

メソッド名	役割
class	例外の種類
message	例外に関するメッセージ
backtrace	例外の発生した位置に関する情報

・ 例えば、ファイルを読み込んでその内容を表示させるといった処理を考えてみます。

以下のようなファイルがあり、これを開いて内容を表示させてみます。

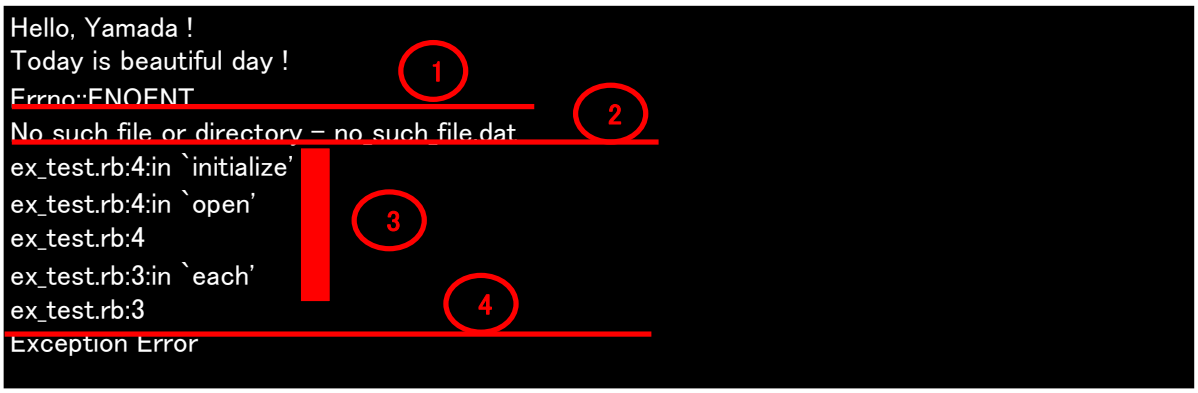
hello.dat

```
Hello, Yamada !
Today is beautiful day !
```

例）

```
begin
    filenames = ['hello.dat', 'no_such_file.dat']
    filenames.each { |filname|
        file = open(filname)
        puts file.read
        file.close
    }
rescue => ex
    puts ex.class          # 例外の種類
    puts ex.message        # 例外メッセージ
    puts ex.backtrace      # 例外発生の位置情報
ensure
    if ex
        puts "Exception Error"
    else
        puts "No Exception Error"
    end
end
```

出力



・ 例）では「hello.dat」と「no_such_file.dat」という二つのファイルを開こうとしています。
（ここで、「hello.dat」は存在するが「no_such_file.dat」は存在しないこととします。）
存在しない「no_such_file.dat」というファイルを開こうとしているので、エラーが生じます。

① ex.classの出力結果です。
② ex.messageの出力結果です。
③ ex.backtraceの出力結果です。
④ 例外の発生の有無に関わらず、メッセージを出力させたい、という意図です。
この例では、例外オブジェクトが得られていますので、真の場合の処理が行われています。

● 正規表現

◆ 正規表現とは

正規表現とは、文字列を抽象的に表現する方法です。正規表現を使うと、
「Rubyのプログラム」や「Javaのプログラム」といった異なる文字列を、

「<半角英文字>のプログラム」

のように1つの形式で表現できます。

◆ メタ文字

正規表現では、**メタ文字** という特殊な意味を持つ文字を使うことが出来ます。

代表的なものには、次のようなものがあります。

文字	持つ意味
^	行頭にマッチします。
$	行末にマッチします。
.	改行を除く任意の1文字にマッチします。
[ ]	[]内のいずれか1文字にマッチします。
[^]	[]内に含まれない1文字にマッチします。
*	直前の表現の0回以上の繰り返しにマッチします。
+	直前の表現の1回以上の繰り返しにマッチします。
?	直前の表現の0回または1回の繰り返しにマッチします。
{n}	直前の表現n回の繰り返しにマッチします。
a|b	aまたはbにマッチします。
(...)	正規表現をグループ化する。マッチしたテキストは記憶されます

```
if /abc/ =~ str
  puts "match"
end
```

出力

```
match
```

この例では、変数strの中に"abc"という文字列が含まれている場合、
matchという文字列を標準出力へ出力します。

このような単純な条件ではなく、
「行頭が#で始まるか？」とか、
「 電話番号が含まれるか？」といった、複雑な形式を表現する場合に、メタ文字を使います。

*こんな使い方ができます。

◆ 繰り返し文字とマッチさせる

◆◆ メタ文字の*、+、?などを使用して繰り返し文字とマッチさせることができます。
*は直前の正規表現の0回以上の繰り返しとマッチします。
以下の例では、直前の文字aが0回以上繰り返しの後にcという文字が存在する場合に
マッチします。

```
/a*c/ =~ "abc" #=> マッチする
/a*c/ =~ "ac" #=> マッチする
/a*c/ =~ "aaaaaac" #=> マッチする
/a*c/ =~ "c" #=> マッチする
/a*c/ =~ "abccccc" #=> マッチする
/a*c/ =~ "abd" #=> マッチしない
```

◆◆ +は直前の正規表現の1回以上の繰り返しとマッチします（最低でも1回以上の繰り返しが必
以下の例では、直前の文字aが1回以上繰り返した後にcという文字が存在する場合に
マッチします。

```
/a+c/ =~ "abc" #=> マッチしない
/a+c/ =~ "ac" #=> マッチする
/a+c/ =~ "aaaaaac" #=> マッチする
/a+c/ =~ "c" #=> マッチしない
/a+c/ =~ "abccccc" #=> マッチしない
/a+c/ =~ "abd" #=> マッチしない
```

◆◆ ?は直前の正規表現の0回、または1回の繰り返しとマッチします。
以下の例では、直前の文字aが0回または1回繰り返した後にcという文字が存在する場合に
マッチします。

```
/a?c/ =~ "abc" #=> マッチする
/a?c/ =~ "ac" #=> マッチする
/a?c/ =~ "aaaaaac" #=> マッチする
/a?c/ =~ "c" #=> マッチする
/a?c/ =~ "abccccc" #=> マッチする
/a?c/ =~ "abd" #=> マッチしない
```

◆ 数字だけ・アルファベットだけとマッチさせる

◆◆ 文字クラスを使用すると数字だけ、アルファベットだけ、AかBかZのどれかなど、
複数文字中の任意の1文字を表現することができます。
文字クラスを指定する場合、複数の文字を[]で括ります。

```
/[ABZ]/ =~ "A" #=> マッチする
/[ABZ]/ =~ "Z" #=> マッチする
/[ABZ]/ =~ "Q" #=> マッチしない
```

◆◆ []内で-を使用すると文字の範囲を示します。
[A-Z]とすればAからZまで、[0-9]とすれば0から9までの数字を表現することができます。

```
/[0-9]/ =~ "5" #=> マッチする
/[0-9]/ =~ "A" #=> マッチしない
/[A-Z]/ =~ "A" #=> マッチする
/[A-Z]/ =~ "5" #=> マッチしない
/[A-Z]/ =~ "a" #=> マッチしない
```

I

◆◆ []内で^を指定すると**「それ以外」**を表現することができます。

```
/[^0-9]/ =~ "A" #=> マッチする
/[^0-9]/ =~ "5" #=> マッチしない
```

◆ n番目のマッチを見つける

・ 正規表現のグルーピングを使用し、
正規表現内のn番目のグループにマッチした文字列という処理を記述することができます。
正規表現内で()で括られた文字列がグループになり、
n番目の要素を参照する場合、$nと記述します。

```
/1(.)3(.)5(.)/ =~ "123456"
puts $1
puts $2
puts $3

puts

/([a-z]+),([¥d¥-]+),(¥S+)/ =~ "take,045-111-2234,Yokohama"

puts $1
puts $2
puts $3
```

出力

```
2
4
6

take
045-111-2234
Yokohama
```

◆ 正規表現を使って文字列を置き換える

・ String#subメソッド、String#gsubメソッドは文字列の置換を行いますが、
置換される文字列の指定に正規表現を使用することができます

```
str = "Hello World".sub(/[A-Za-z]*ld/, "Yamada")

puts str
```

出力

```
Hello Yamada
```

◆ マッチした文字列を全て抜き出して配列へ格納する

・ String#scanメソッドは文字列中で任意のパターンにマッチするものを全て抜き出し、
配列へ格納することができます。

```
p "hoge:045-111-2222 fuga:045-222-2222".scan(/(¥S+):([¥d¥-]+)/)
```

出力

```
[["hoge", "045-111-2222"], ["fuga", "045-222-2222"]]
```

● メソッド

メソッドとは、一般にいうところの「関数」です。
Rubyのようなオブジェクト指向のプログラミング言語においては、クラスを元にオブジェクトを作成し、
クラスの中に定義されたメソッドをオブジェクトから呼び出します。

◆ メソッドの呼び出し

・ メソッド呼び出しの一般的な構文は、次のようになります。

```
オブジェクト.メソッド名(引数1, 引数2, 引数3, …, 引数n)
```

先頭にオブジェクトが一つ置かれ、その後ろにピリオド「.」を挟み、メソッド名が続きます。
メソッド名の後ろには「()」で囲まれたメソッド引数が並びます。
なお、この「()」は省略できます。

◆ レシーバ

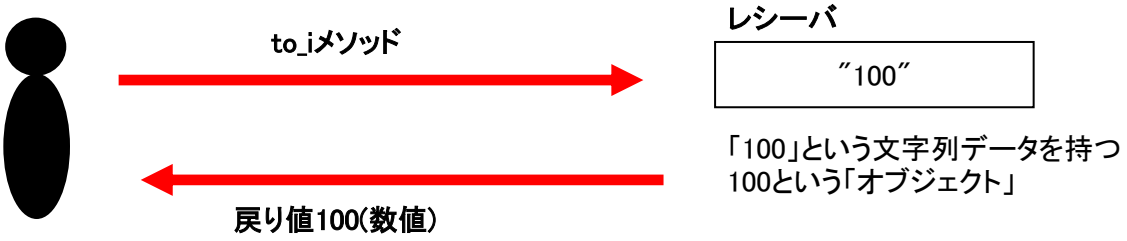
上の構文において、オブジェクトのことを「レシーバ」と呼びます。

```
オブジェクト.メソッド名(引数1, 引数2, 引数3, …, 引数n)
```

これは、オブジェクト指向の世界では、メソッドを実行することを
「オブジェクトにメソッド(メッセージ)を送る」「オブジェクトはメソッドを受け取る(receiveする)」と
解釈するためです。

つまり、あるオブジェクトに対し、いくつかの引数とともにメッセージが送られる、
というイメージです。

```
n = "100".to_i
```



● メソッドの分類

◆ Rubyのメソッドはレシーバのタイプによって3種類に分けることが出来ます。

◆ **インスタンスメソッド**
もっとも一般的なメソッドです。あるオブジェクト(インスタンス)があったとき、
そのオブジェクトをレシーバとするメソッドのことをインスタンスメソッドと呼びます。

```
"10, 20, 30, 40".split(", ")
[1, 2, 3, 4].index(2)
1000.integer?
```

上から順に、文字列、配列、数値の各オブジェクトがレシーバとなります。

＊ integer? メソッドは、数値自身が整数かどうかを判定します。値が整数のときはtrue、
整数以外のときはfalseが返ります。

◆ **クラスメソッド**
レシーバがインスタンスではなく、クラスそのものだった場合、そのメソッドは
クラスメソッドと呼ばれます。
インスタンスを生成したりするときには、クラスメソッドが使われます。

```
a = Array.new
f = File.open("some_file")
t = Time.now
```

上から順に、
・ 新しい配列を作る
・ 新しいファイルオブジェクトを作る
・ 新しい時刻オブジェクトを作る

といった処理です。

◆ **関数的メソッド**
レシーバが無いメソッドを関数的メソッドと言います。

```
sin(3.14)
sleep(10)
print "hello!"
```

上から
・ sin()関数
・ 指定された秒数、処理を停止する
・ コンソールに文字列を出力する

という処理です。

● メソッドの定義

◆ メソッド定義の一般的な構文は次のようなものです。

```
def メソッド名(引数1, 引数2, …)
    実行したい処理
end
```

- メソッド名には**アルファベット、数字、アンダースコア(_)**を使用することが出来ます。
  ただし、**数字で始めることはできません。**

◆ デフォルト値の指定

- 引数を省略してメソッドを呼び出した場合、デフォルトの値が渡されます。

  （引数名 = 値）

  という書き方をします。

```
def hello(name = "Ruby")
  print("Hello, ", name, ".¥n")
end

hello()                    #=>引数無しで呼び出す
hello("Newbie")            #=>引数を指定して呼び出す
```

出力

```
Hello, Ruby.
Hello, Newbie
```

- メソッドが複数の引数を持つ場合は、引数リストの**右端から順に**デフォルト値を設定します。
  例えば、3つの引数のうち2つを省略可能にする場合は、右側の二つに値をセットするようにします

```
def func(a, b = 1, c = 2)
 .
 .
 .
end
```

左端の引数や、あるいは途中の引数だけを省略可能にすることはできません。

● メソッドの返り値

◆ メソッドの中でreturn文を用いることで、メソッドの返り値を指定できます。

```
return 値
```

例）

```
def volume(x, y, z)
  return x*y*z
end

p volume(2, 3 ,4)
p volume(10, 20 ,30)
```

出力

```
24
6000
```

例では、立方体の面積を求め、その結果を返しています。

- return文は**省略可能**です。

```
def volume(x, y, z)
  x*y*z
end

p volume(2, 3 ,4)
p volume(10, 20 ,30)
```

出力

```
24
6000
```

● スコープ

◆ Rubyでは、変数名や定数名で、それらの持つスコープが分かるようになっています。

対象	記法	範囲
ローカル変数	小文字、またはアンダースコア[_]で始まる	メソッド内など、特定の範囲でのみ使用可能な変数
インスタンス変数	アットマーク1つ[@]で始まる	クラスに定義されたインスタンスメソッド内で使用可能な変数
クラス変数	アットマーク2つ[@@]で始まる	クラス内であればどこでも使用可能な変数
グローバル変数	ドル記号[$]で始まる	スクリプトのどこからでも使用可能な変数
定数	アルファベット大文字	定義された位置により異なる クラス内に定義：： ・クラス内からは普通に参照可能。 ・クラス外からはスコープ演算子[：：]を付けて参照可能 クラスの外に定義：： ・修飾子を付けずに、またはスコープ演算子だけで参照可能

● エイリアス

◆ Rubyでは、既に定義されているメソッド・演算子・グローバル変数の別名を定義することが出来ます。

```
alias 新しく定義する名前　既存に定義されていた名前
```

例えば、「ten」という名前で定義されているメソッドを、「ju」という名前で定義しなおすとすると・・・

```
def ten
  return 10
end

alias ju ten

ju      #=>10
ten     #=>10
```

名前が変更されただけなので、**処理内容、出力結果などは同じです。**

● シンボル

◆ シンボル(Symbol)というのは、「文字列と1対1に対応する値」を表すクラスです。
Rubyの内部で、メソッド名などの識別に使われているのは、文字列ではなくシンボルです。

■ ある名前のシンボルを作成するには、その名前の前にコロン[:]を付けます。

```
a = :"dog"
b = :"dog"
c = :"dog"
d = :"cat"
```

■シンボルの値を得るには、to_iメソッドで変換してやります。
ここで…

変数aに代入された
:"dog"
の値
=28801

変数bに代入された
:"dog"
の値
=28801

変数cに代入された
:"dog"
の値
=28801

変数dに代入された
:"cat"
の値
=28861

上記のように、シンボルの名前が同じであればその値も同じとなります。

◆ オブジェクトIDとシンボルIDは別物です

■ 対象のオブジェクトのオブジェクトIDを得るには、object_idメソッドを使用します。
同じ名前のシンボルであれば、オブジェクトIDも同じになります。

変数aに代入された
:"dog"
のオブジェクトIDの値
=288018

変数bに代入された
:"dog"
のオブジェクトIDの値
=288018

変数cに代入された
:"dog"
のオブジェクトIDの値
=288018

変数dに代入された
:"cat"
のオブジェクトIDの値
=288098

■ シンボルに対して、to_sメソッドを使用すれば、文字列として得ることができます。

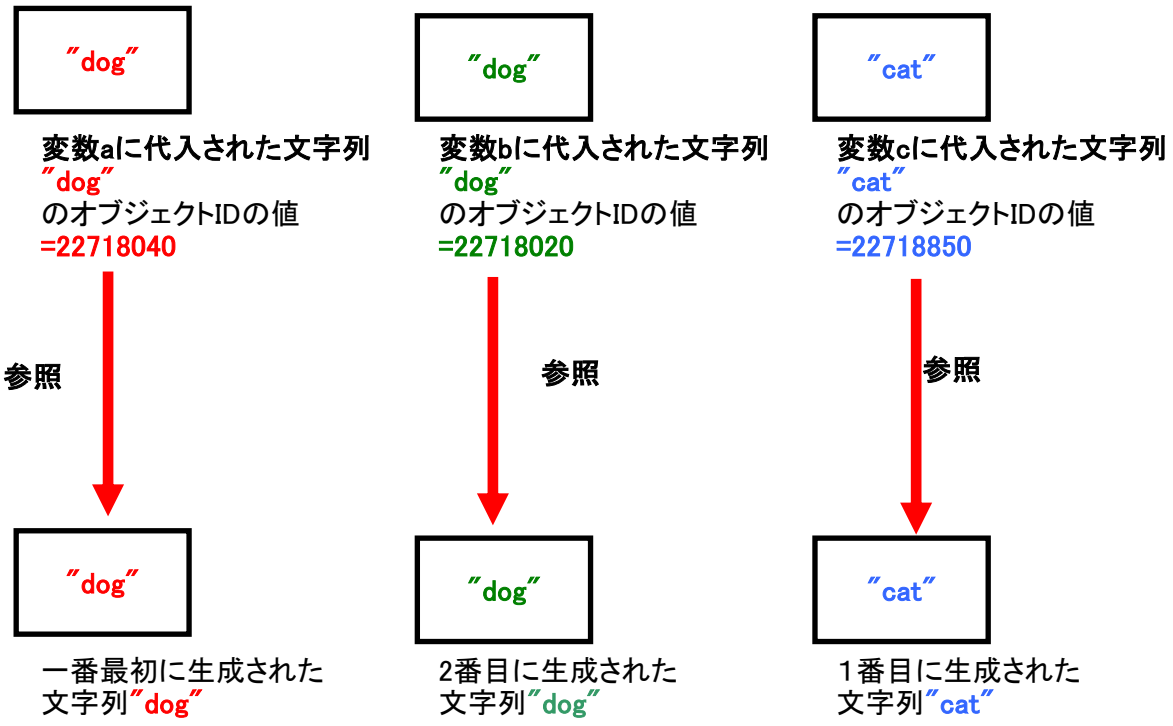
:"dog".to_s　　→　　"dog"
:"cat".to_s　　→　　"cat"

● シンボルの使い道

◆ メモリの節約

・ 例えば、同じ文字列を大量に繰り返し表示したい・・・などと言った場合、
文字列オブジェクトであれば、全く同じ文字列であったとしても、生成されるものは別物です

```
a = "dog"
b = "dog"
c = "cat"
```

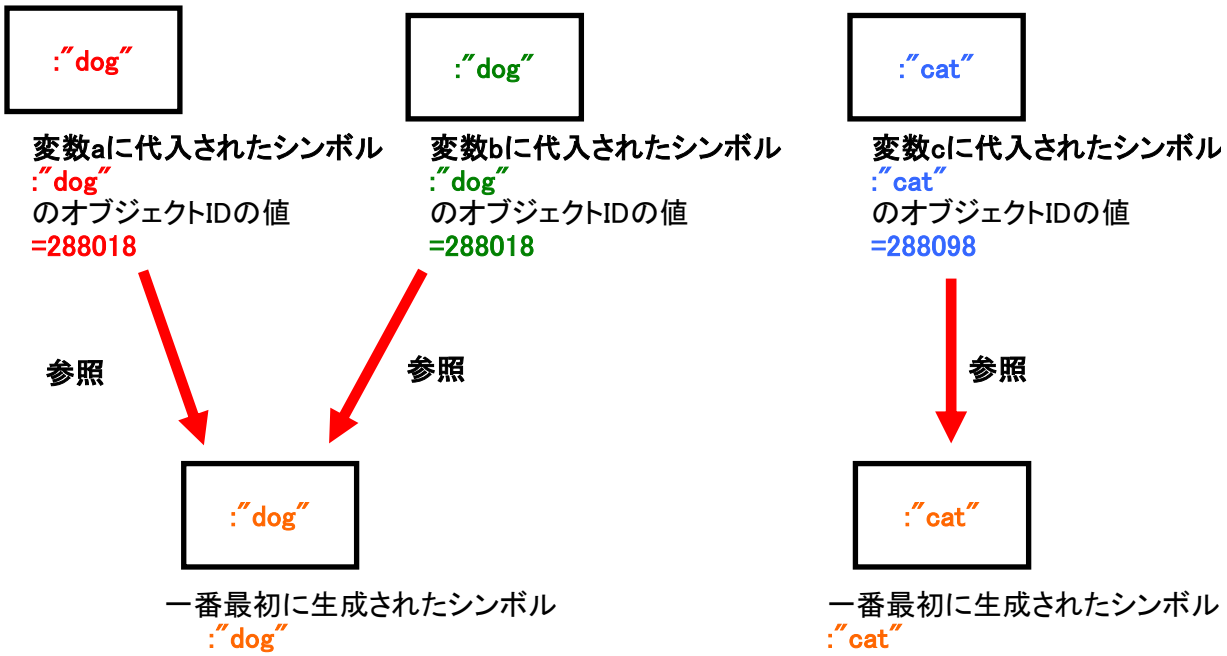


生成されればされるだけ、それだけメモリを消費することになります。

・ しかしシンボルの場合は、全く同じ名前であれば、**参照するものは同じです。**

```
a = :"dog"
b = :"dog"
c = :"cat"
```

シンボルは一度生成されてしまえば、名前が同じなら全て同じものを参照しています。



シンボルは、to_sメソッドを使用すれば、**文字列の代わりとして使用できます。**
↓
つまり、パフォーマンスの向上につながります。

◆ **可読性の向上**
■ シンボルは、ハッシュのキーとして使われることが多いです。

配列で下記のような表記をするものを、

```
array = [value1, value2, value3]                    #=># array[0]→"value1"
```

ハッシュを用いれば下記のように表記でき、

```
hash = {"key1"=>"value1", "key2"=>"value2", "key3"=>"value3"}    #=> hash["key1"]→"value1"
```

これを、シンボルを用いて書き直すと下記のようになります。

```
hash = {key1 : "value1",  key2 : "value2",  key3: "value3" }        #=>hash[:key1]→"value1"
```

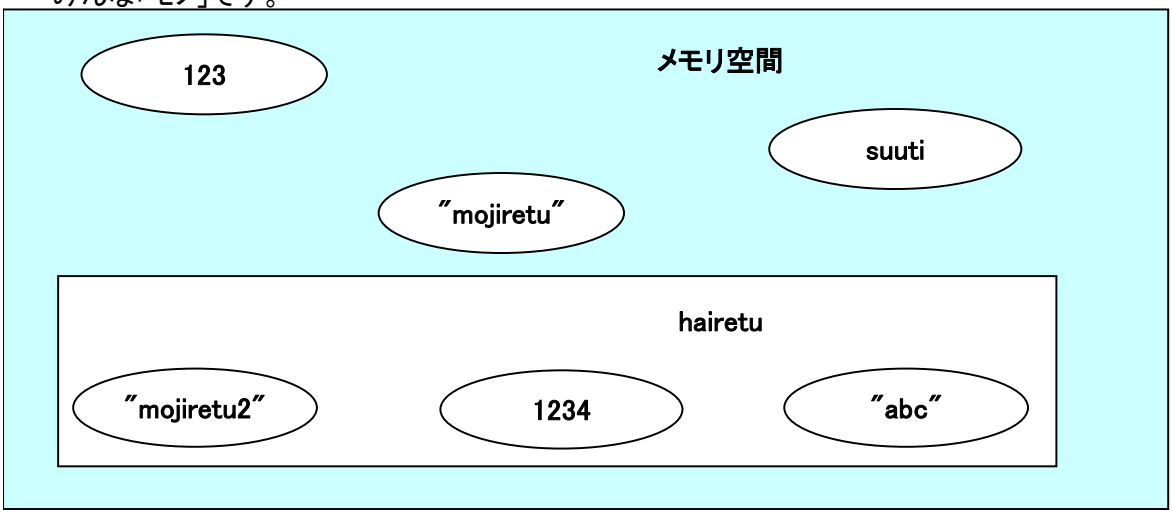
ダブルクォーテーションの表記が少なくなり、
加えて、文字列ではなくシンボルを使うことで何かの名前を参照しているということが明確になるので、
可読性が向上します。

◆ **文字列そのものに意味があるのか、データに意味があるのか**
考え方としては、列挙型 enum に近いものがあるようです。

● オブジェクト

◆ Rubyでは、全てのデータをオブジェクトとして扱います。オブジェクトは「モノ」です。
オブジェクトは、メモリ上に生成されたデータ単位です。
数値も文字列も配列も、それぞれのデータの塊としてメモリ上に保持された「モノ」です。

みんな「モノ」です。



データを、「性質と振る舞いを持つモノ」として捕らえれば、管理や操作の対象として捕らえやすくなります。

■ Rubyのオブジェクトの「型」には、次のようなものがあります。

オブジェクト	説明
数値型オブジェクト	数を表します。
文字列型オブジェクト	文字列を表します。
正規表現オブジェクト	文字列のマッチングのためのパターンを表します
時刻オブジェクト	時刻を表します。
ファイルオブジェクト	ファイルへの読み書きを行うのに用いられます。
配列型オブジェクト	配列を表します。
ハッシュ型オブジェクト	ハッシュを表します。

● クラス

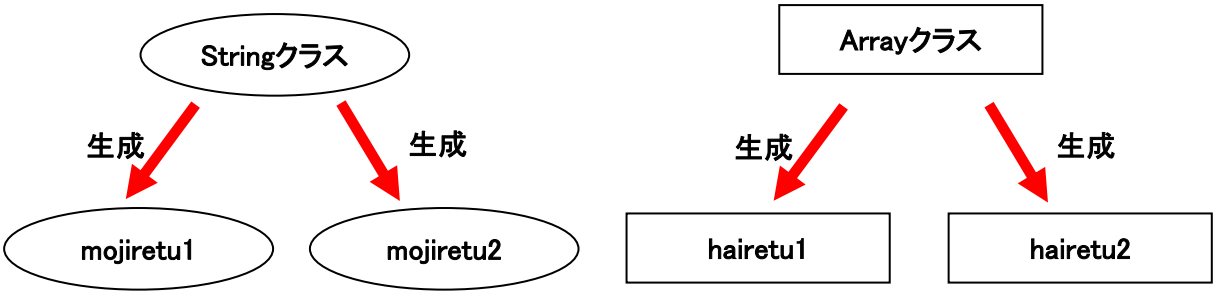
◆ クラスは、オブジェクトの種類を表すものです。Rubyのオブジェクトは、
全てが何らかのクラスに属しています。

クラス	クラス名（classﾒｿｯﾄﾞ実行）
Object	Object
文字列	String
シンボル	Symbol
整数	Fixnum, Bignum
浮動小数点数	Float
配列	Array
ハッシュ	Hash
正規表現	Regexp
範囲	Range

classメソッドを使用すれば、
そのオブジェクトの属する
クラスが分かります。

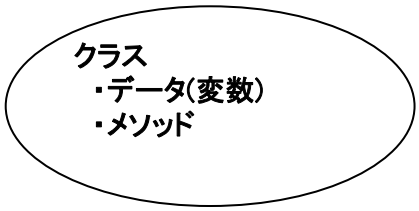
```
self.class
"hoge".class
:hoge.class
10.class
```

◆ クラスは、オブジェクトを生成するための「型」です。
オブジェクトの振る舞いを決める、雛形、あるいは設計図と表現されます。



Stringクラスから生成されたオブジェクトmojiretu1とmojiretu2は
同じ振る舞いをします。Arrayクラスから生成されたオブジェクトhairetu1とhairetu2についても、
同様です。

◆ クラスは、データとメソッドを持ちます。



● クラスを作る

◆ クラスを作るには、class文を用いて定義する必要があります。

```
class  クラス名
      クラスの定義
end
```

■ クラス名は、**必ず英大文字で始めなければなりません。**
クラス名として2つ以上の単語が並ぶ場合、大文字と小文字を混在させて、
各単語の先頭文字を大文字にするのが慣例のようです。

```
class  ClassName
      クラスの定義
end
```

◆ **インスタンス変数とクラス変数**
クラスに実装できる変数に、インスタンス変数とクラス変数があります。

◆◆ **インスタンス変数**
■ クラス内で、@で始まる変数をインスタンス変数といいます。次のような性質があります。
・ メソッド内で定義され、そのメソッドを抜けても値は保持される。
・ インスタンスごとに違った値を保持できる。
・ 外部からの**直接**の参照や代入が出来ない。そのために、アクセスメソッドを
クラス内に用意する必要がある。
・ インスタンスメソッドからの参照は可能。

◆◆ **クラス変数**
■ @@で始まる変数がクラス変数です。性質は以下のようなものです。
・ **そのクラス全てのインスタンスで共有できる変数。**
・ 定数に似ているが、何度でも値を変更できる。
・ 参照するにはアクセスメソッドが必要。

◆ クラスのメソッド

■ メソッドは実行する一連の処理をまとめたものです。クラスにメソッドを実装するには、クラス内に定義する必要があります。

・ メソッドの定義には、defを使用します。

```
class クラス名
    def メソッド名(引数1, 引数2, …引数n)
        実行したい処理
    end
end
```

・ メソッド名は英小文字で始めます（末尾に ？， ！， = の3つが使用可能）
・ メソッドの引数の括弧は省略可能です。

```
3.to_s(2)        #=>2進数に変換
3.to_s 2         #=>2進数に変換
```

＊ ただし、紛らわしいので、引数のあるメソッドは括弧を付け、引数のないメソッドは括弧を省略するスタイルをお勧めします。

■ メソッドには、インスタンスメソッド、アクセスメソッド、クラスメソッドがあります。

◆◆ インスタンスメソッド

・ 通常のメソッドがプログラム中からいつでも呼び出せるのに対して、クラス内に記述されたメソッドはクラスから作成されたオブジェクトしか呼び出すことが出来ません。このようなメソッドをインスタンスメソッドと呼びます。もっとも一般的なメソッドです。

```
class A
  def method1
      実行したい処理
  end
end
```

生成

```
a = A.new
```

Aクラスのインスタンス
a

```
a.method1
```

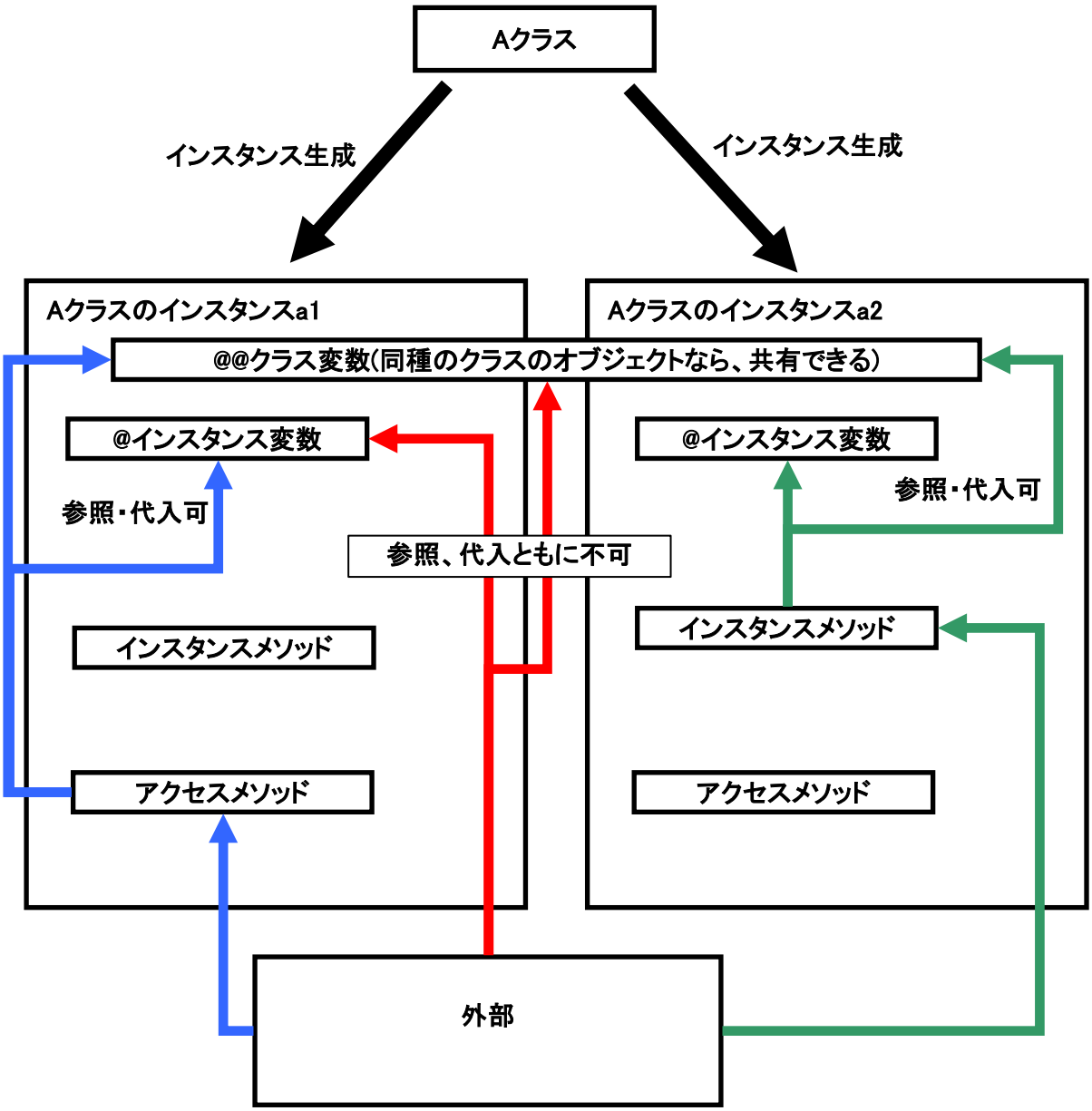
インスタンスが生成されることで、クラス内に定義されたメソッドが使えるようになります。

サンプルプログラム

```
class Song
  @@plays = 0  # ｸﾗｽ変数 同じｸﾗｽのｲﾝｽﾀﾝｽ同士で共有可能
  def initialize(name, artist, duration) # ｲﾝｽﾀﾝｽ初期化ﾒｿｯﾄﾞ newで呼ばれる
    @name      = name      # ｲﾝｽﾀﾝｽ変数 ｲﾝｽﾀﾝｽ毎に
    @artist    = artist    # ｲﾝｽﾀﾝｽ変数
    @duration  = duration  # ｲﾝｽﾀﾝｽ変数
    @plays     = 0         # ｲﾝｽﾀﾝｽ変数 ｲﾝｽﾀﾝｽ毎に値を保持。ﾒｿｯﾄﾞを抜けても値を保持。
  end
  def play     # ｲﾝｽﾀﾝｽﾒｿｯﾄﾞ
    @plays+=1  # ｲﾝｽﾀﾝｽ変数
    @@plays+=1 # ｸﾗｽ変数
    p "This song #{@plays} plays. Total #{@@plays} plays."
  end
end

song1=Song.new("aaa","bbb",123) # Songｸﾗｽのｲﾝｽﾀﾝｽ生成,初期化
song2=Song.new("xxx","yyy",456) # Songｸﾗｽのｲﾝｽﾀﾝｽ生成,初期化
song1.play # 1曲目 song1x1回目
song1.play # 2曲目 song1x2回目
song2.play # 3曲目 song2x1回目
song1.play # 4曲目 song1x3回目
song2.play # 5曲目 song2x2回目
song1.play # 6曲目 song1x4回目
song2.play # 7曲目 song2x3回目
song1.play # 8曲目 song1x5回目
song2.play # 9曲目 song2x4回目
```

実行結果

```
"This song 1 plays. Total 1 plays."
"This song 2 plays. Total 2 plays."
"This song 1 plays. Total 3 plays."
"This song 3 plays. Total 4 plays."
"This song 2 plays. Total 5 plays."
"This song 4 plays. Total 6 plays."
"This song 3 plays. Total 7 plays."
"This song 5 plays. Total 8 plays."
"This song 4 plays. Total 9 plays."
```



- クラスのインスタンスメソッドを介せば、クラス変数とインスタンス変数への参照・代入が可能で
- アクセスメソッドを介せば、インスタンスメソッドを定義する手間が省けます。

◆◆　initializeメソッド
初期化処理を行うメソッドです。

```
class Car
  def initialize(carname="未定義")
    @name = carname
  end

  attr_accessor :name
end

car = Car.new("PRIUS")
p car.name
```

```
"PRIUS"
```

この例では、Carクラスのインスタンスcarを生成する際、「PRIUS」で@nameを初期化しています。
ですが、initializeメソッドにはデフォルト値で「未定義」という値が指定されていますので、

```
car = Car.new
```

と引数無しで初期化してやった場合は

```
"未定義"
```

自動的に"未定義"が渡されます。

◆◆　アクセスメソッド

・　クラスの中で使われているインスタンス変数はクラスの外からは直接
参照したり値を変更したりすることが出来ません。参照する場合も変更したい場合も
インスタンスメソッドを経由して行う必要があります。
しかし、インスタンス変数を大量に保持するようなクラスの場合、インスタンスメソッドを
定義するだけでも大変になります。

```
class Car
  def initialize(carname="未定義")
    @name = carname
  end
  def name # ｲﾝｽﾀﾝｽ変数を参照するためにｲﾝｽﾀﾝｽﾒｿｯﾄﾞを定義
    @name
  end
end

car = Car.new("PRIUS")
p car.name
```

そこでインスタンス変数への参照や更新が簡易的に行えるように
アクセスメソッドと呼ばれるものが用意されています

定義式	機能
attr_reader :**変数名**	参照が可能
attr_writer :**変数名**	更新が可能
attr_accessor :**変数名**	参照と更新が可能

```
class Car
  def initialize(carname="未定義")
    @name = carname
  end
  attr_accessor :name
end

car = Car.new("PRIUS")
p car.name
```

上記のように対象となるインスタンス変数名に対して「attr_reader」「attr_writer」「attr_accessor」
のいずれかを使って上記のように記述することでインスタンス変数の参照や更新用のメソッドを
個別に定義する代わりとなります。

attr_accessor　を使用していますので、上記のように更新と参照が
両方行えることになります。

アクセスメソッドは、「アクセサ」とも呼ばれます。

◆◆ クラスメソッド

Rubyでは、クラスもオブジェクトとして扱います。クラスを定義することは、
そのクラスそのものをインスタンスとして生成していることになります。

```
class MyClass
  def method
    print "a"
  end
end
```

MyClass というクラスオブジェクトを生成したことになります。

クラスメソッドは、あるクラスオブジェクトに対して定義できる、
そのクラスオブジェクト固有のメソッドです。

クラスメソッドの定義方法

```
class MyClass
  def method
    print "method"
  end
  def self.classMethod  # ここは MyClass.classMethod でも定義可能ですが、クラス名を変更
    print "classMethod" # する場合に変更箇所が複数にならないので self の方が便利です。
  end
end
```

```
def MyClass.classMethod  # クラスの外でも定義が可能
  print "classMethod"
end
```

特異クラス定義(特定のオブジェクトに対するメソッドの定義)によるクラスメソッドの定義

```
class MyClass
  def method
    print "method"
  end
  class <<self  # ここは MyClass でも定義可能
    def classMethod
      print "classMethod"
    end
  end
end
```

```
class <<MyClass  # クラスの外でも定義が可能
  def classMethod
    print "classMethod"
  end
end
```

◆ 継承

■ Rubyにも継承機能があります。
ただし、**クラスは自分のすぐ上の親を1つしか持てません**（単一継承）。

継承を行うには、class文で指定するクラス名と同時にスーパークラス名を指定します。

```
class クラス名 ＜ スーパークラス名

        追加したいクラス定義
end
```

◆ オーバーライド

■ クラスの継承において、スーパークラスで定義されているメソッドを
サブクラスから定義しなおすことをオーバーライドといいます。

```
class Car
  def accele
    print("アクセルを踏みました¥n")
  end
  def brake
    print("ブレーキを踏みました¥n")
  end
end

class Soarer < Car
  def openRoof
    print("ルーフを開けました¥n")
  end
  def accele
    print("アクセルを踏んで加速しました¥n")
  end
  def brake
    print("ブレーキを強く踏みました¥n")
  end
end

class Crown < Car
  def reclining
    print("シートをリクライニングしました¥n")
  end
end

car = Car.new
car.accele
car.brake

soarer = Soarer.new
soarer.accele
soarer.brake
soarer.openRoof

crown = Crown.new
crown.accele
crown.brake
crown.reclining
```

◆ 継承

出力

```
アクセルを踏みました
ブレーキを踏みました
アクセルを踏んで加速しました
ブレーキを強く踏みました
ルーフを開けました
アクセルを踏みました
ブレーキを踏みました
シートをリクライニングしました
```

- **赤い記述**が継承で追加した処理、**青い記述**がオーバーライドで変更した処理です。
  サブクラスでオーバーライドされなかった部分は、スーパークラスのものが引き継がれます。

◆ **super**を使えば、そのクラスの親クラスの同名のメソッドを呼び出すことが出来ます。

```
class Car
  def accele
    print("アクセルを踏みました¥n")
  end
end

class Soarer < Car
  def accele
    super
    print("加速しました¥n")
  end
end

soarer = Soarer.new
soarer.accele
```

出力

```
アクセルを踏みました
加速しました
```

- super によって、Carクラスのaccele が呼び出されています。

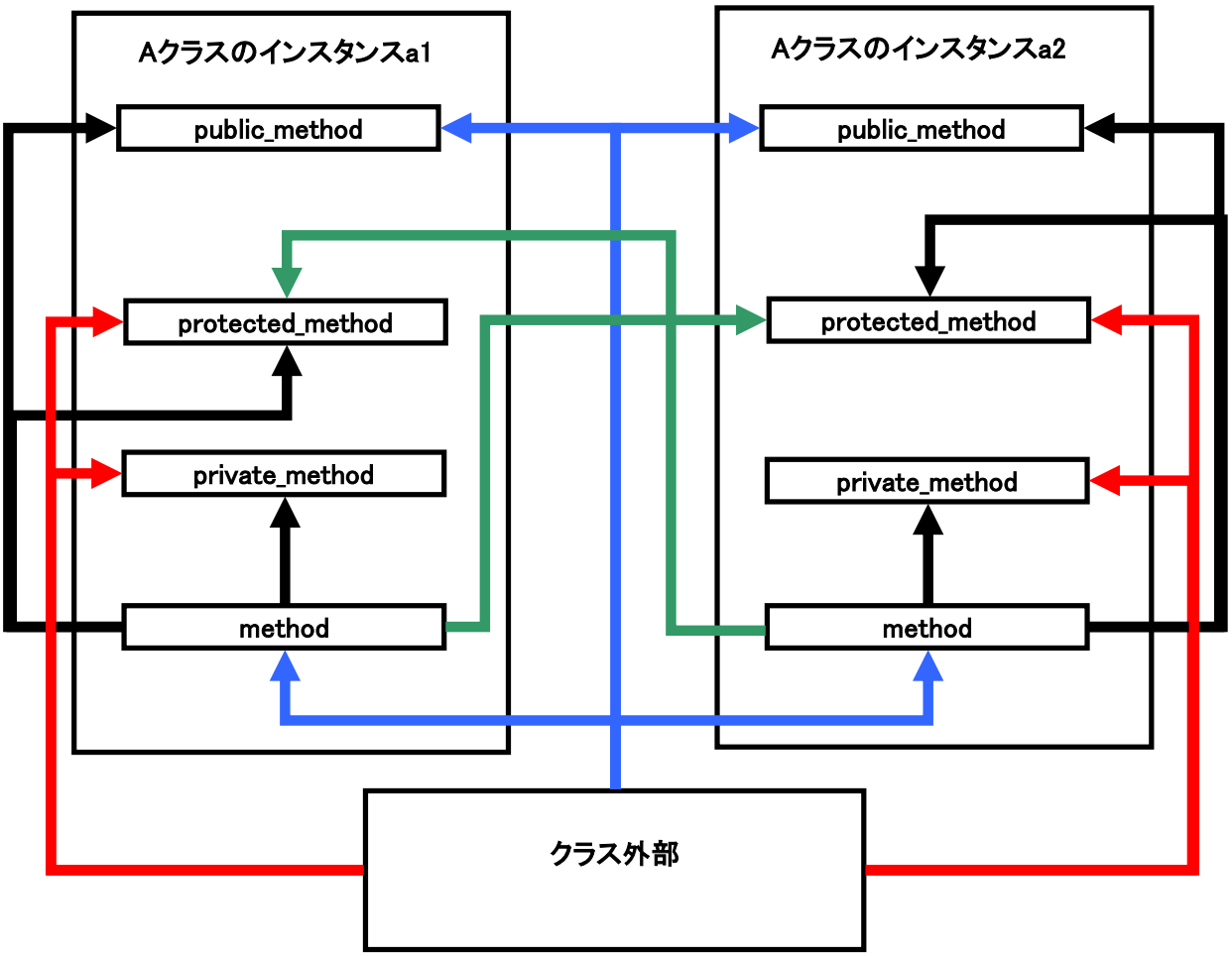
super の 呼び出し方法

super	現在のメソッドの引数をそのままスーパークラスのメソッドに渡す
super()	引数なしでスーパークラスを呼び出す
super(arg)	指定した引数をスーパークラスのメソッドに渡す

◆ **アクセス修飾子**

■ メソッドに対して、アクセス制限を付加したいときに使用します。
アクセス制限のレベルは3段階あります。

publicメソッド	誰でも呼び出せるメソッド アクセス制御は行わない
protectedメソッド	定義元クラスおよびサブクラスのオブジェクトだけが呼び出せる。 それ以外のオブジェクトによるアクセスは禁止
privateメソッド	現在のオブジェクト内でしか呼び出せない 関数形式でのみ呼び出せる



- 外部からのpublicメソッド及び無修飾インスタンスメソッドへのアクセスは可能
（インスタンスメソッドはデフォルトでpublicとなる）
- 外部からのprivateメソッド及びprotectedメソッドへのアクセスは不可能。
- おなじクラスのインスタンスのメソッドであれば、他の同種のインスタンスのprotectedメソッドへのアクセスは可能。
- 無修飾インスタンスメソッドは、あらゆるメソッドにアクセス可能。

インスタンスメソッド用に、public、protected、private
クラスメソッド用に、public_class_method、private_class_method があります。

サンプルプログラム

```ruby
class MyClass
  def method  # 何も記述しないと public です。
    puts "method"
  end
 protected    # これ以降 protected です。
  def prot
    puts "protected_method"
  end
 private      # これ以降 private です。
  def priv
    puts "private_method"
  end
 public       # これ以降 public です。
  def publ
    puts "public_method"
  end
  def print_all
  # 同じｸﾗｽのｲﾝｽﾀﾝｽﾒｿｯﾄﾞからｱｸｾｽ可能
    self.method # 関数的･ｲﾝｽﾀﾝｽﾒｿｯﾄﾞでｱｸｾｽ可能
    self.prot   # 関数的･ｲﾝｽﾀﾝｽﾒｿｯﾄﾞでｱｸｾｽ可能
    priv        # 関数的ﾒｿｯﾄﾞのみｱｸｾｽ可能
    self.publ   # 関数的･ｲﾝｽﾀﾝｽﾒｿｯﾄﾞでｱｸｾｽ可能
  end
end

class SubMyClass < MyClass # MyClass がｽｰﾊﾟｰｸﾗｽ
  def sub_met
    method      # ｻﾌﾞｸﾗｽからｱｸｾｽ可能
  end
  def sub_prot
    prot        # ｻﾌﾞｸﾗｽからｱｸｾｽ可能
  end
  def sub_priv
    priv        # ｻﾌﾞｸﾗｽからｱｸｾｽ可能
  end
  def sub_publ
    publ        # ｻﾌﾞｸﾗｽからｱｸｾｽ可能
  end
end

a=SubMyClass.new
a.sub_met
a.sub_prot
a.sub_priv
a.sub_publ

print "¥n"

b=MyClass.new
b.print_all
b.method
b.publ
b.prot # Error : ｸﾗｽ外部からのｱｸｾｽ不可
b.priv # Error : ｸﾗｽ外部からのｱｸｾｽ不可
```

```ruby
class MyDate
  def initialize(year,month,day)
    @year=year
    @month=month
    @day=day
  end
  def <=>(target)
    # 同じｸﾗｽのｲﾝｽﾀﾝｽからｱｸｾｽ可
    ret = @year  - target.year
    ret = @month - target.month if ret==0
    ret = @day   - target.day   if ret==0
    ret
  end

# public     # OK
  protected  # OK
# private    # Error
  def year
    @year
  end
  def month
    @month
  end
  def day
    @day
  end
end


date1=MyDate.new(2009,1,10)
date2=MyDate.new(2009,1,15)
p date1 <=> date2

# => 実行結果 -5
```

=>

```
method
protected_method
private_method
public_method

method
protected_method
private_method
public_method
method
public_method
bbbbbb.rb:52: protected method `prot' c
rror)
```

● モジュール

◆ モジュールとは

■ クラスは実体(データ)と振る舞い(処理)を持った「モノ」を表現するためのものでしたが、
モジュールは処理の部分だけをまとめるものです。

- モジュールはインスタンスを生成できない
- モジュールは継承できない

といった特徴があります。

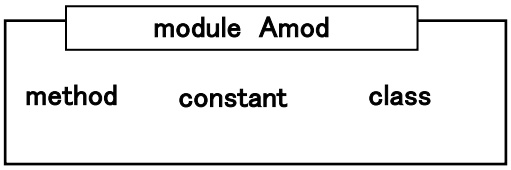
次のように定義します。

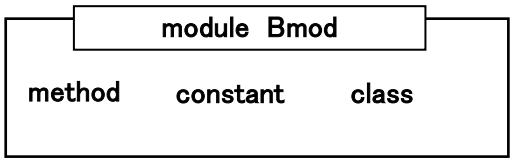
```
module　モジュール名
　　モジュールの定義
end
```

- キーワード module を使って定義します。
- モジュール名の1文字目は大文字で始めます。
- メソッド定義の1文字目は小文字で始めます。
- モジュールが他のファイルに記述されている場合は、まずrequireを使って
ファイルを読み込む必要があります。

◆ 名前空間としてのモジュールの利用

- 同じ定数名やメソッド名でも、違う名前空間に属していれば違うものとして扱われます。
モジュールを利用して、定数名やメソッド名の衝突を防ぐことが出来ます。





◆ 定義と呼び出し

モジュールに定義したメソッドを使用するには、モジュール内でmodule_functionを使って
モジュール関数として呼び出せるようにします。
関数の呼び出しは

```
モジュール名.メソッド名
```

定数の呼び出しは

```
モジュール名::定数名
```

で行います。

```ruby
module A
  Version = "1.0"
  def hello(name)
    print("Hello, ",  name, "¥n")
  end
  module_function :hello
end

module B
  Version = "2.0"
  def hello(name)
    print("こんにちは, ",  name, "さん", "¥n")
  end
  module_function :hello
end

p A::Version
A.hello("Yamada")

include A
p Version
hello("Saito")

include B
p Version
hello("Saito")
```

出力

```
 "1.0"
Hello, Yamada
"1.0"
Hello, Saito
"2.0"
こんにちは, Saitoさん
```

A::Version でモジュールAの定数Versionを呼び出し、A.hello("Yamada")で
モジュールAのメソッドhelloを呼び出しています。

◆ include

- includeを用いれば、指定したモジュールを読み込むことが出来ます。
  モジュールを読み込んだ後なら、**定数やメソッドの呼び出し時のモジュール名は省略できます**
  しかし、指定したモジュールを読み込んだ以降は、そのモジュールが適用されますので、上記
  うに複数のモジュールが存在する場合はモジュール名を省略しないほうが何かと便利です。

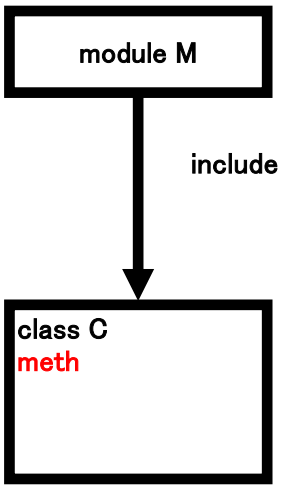
● **クラスへのモジュール読み込み**
  ◆ なお、クラス定義の中でincludeを使うと、モジュールに含まれるメソッドや定数を
  クラスの中に取り込むことが出来ます。これをMix_inといいます。

```
module M
  def meth
    puts "meth"
  end
end

class C
  include M
end

c = C.new
c.meth
```

出力

```
meth
```

モジュールに定義されたメソッドmethが、クラスCの中に取り込まれています。



モジュールがクラスのスーパークラスになります。Rubyは単一継承ではありますが、
このMix-inにより多重継承が可能になります。

◆ **標準ライブラリ**

Ruby言語に標準で添付されているライブラリを使用するには、
requireで呼び出します。

```
require　"使いたいライブラリのファイル名"
```

◆ **csv**
- csvファイルを取り扱います。

◆ **date**
- 日付を扱うためのライブラリです。

◆ **debug**
- Ruby デバッガです。Ruby スクリプトのソースコードデバッグに使用します

◆ **Win32API**
- Win32 API を呼び出すためのライブラリです。

◆ **find**
- ディレクトリ配下のファイルを探索するためのモジュールです。

なお、今までに紹介したArrayクラスやStringクラスといったものは、
Rubyの中に組み込まれたライブラリから使用されますので、使用する際にrequireを
記述する必要はありません。

● イテレータ

◆ イテレータとは

- イテレータは、Rubyの特徴的な機能の一つです。繰り返し処理です。

- Rubyのループ処理は、単純なものしか用意されていません。イテレータを使用すれば、
  より強力なループ処理を行うことが出来ます。
  例えば、配列の要素数を調べ、その範囲でループ処理を行えます。
  そのため、開発効率が格段に上がる可能性があります。

いくつか例をあげてみます。

◇ each

代表的な繰り返し処理です。

```
[1, 2, 3].each{| x | puts x*x }
```

出力

```
1
4
9
```

範囲演算子を使用することも出来ます。

```
(1..3).each{| x | puts x*x }
```

出力

```
1
4
9
```

◇ **times**
繰り返し範囲が明確な場合に使われます。

```
5.times{ puts "繰り返し"}
```

出力

```
繰り返し
繰り返し
繰り返し
繰り返し
繰り返し
```

◇ **select**
ある条件を満たす要素のみの配列を返します。
次の例では、整数のみのリストを取得します。

```
[1.1, 2, 3.3, 4].select{| x | x.integer?}
```

出力

```
2
4
```

◇ **reject**
selectの反対で、ある条件を満たさない要素の配列を返します。
次の例では、整数以外のリストを取得します。

```
[1.1, 2, 3.3, 4].reject{| x | x.integer?}
```

出力

```
1.1
3.3
```

◇ **collect**
新しい要素の配列を、返します。
次の例では、文字列の配列から、各要素の長さの配列を生成します。

```
a = []
i = 0
["a", "aa", "aaa", "b", "cc"].collect{| x |  a[i] = x.size
i+=1}

a.each{|y| puts y}
```

出力

```
1
2
3
1
2
```

◇ find
条件に一致する要素を取得します。
次の例では、文字列の配列から先頭が「b」に一致するものを取得します。

```
puts ["a", "aa", "aaa", "b", "cc"].find{| x | a = (/^b/ =~ x)}
```

出力

```
b
```

◇ sort{| a,b | ...}
並び順の評価式を指定してソートします。
次の例では、配列の要素を文字順、数値順、降順でソートします。

```
puts ["3", "5", "14", "2", "1", "10"].sort
```

出力(文字順)

```
1
10
14
2
3
5
```

```
puts ["3", "5", "14", "2", "1", "10"].sort{ | a, b | a.to_i <=> b.to_i }
```

出力(数値順)

```
1
2
3
5
10
14
```

```
puts ["3", "5", "14", "2", "1", "10"].sort{ | b, a | a.to_i <=> b.to_i }
```

出力(降順)

```
14
10
5
3
2
1
```

● イテレータ
◆ ブロックを使用する

◇ ブロック
「メソッド呼び出し時に追加できるコードの塊」です。

■ { ～ }で囲む

```
{
  繰り返したい処理
}
```

{ ～ } で囲んだ部分が、ブロックになります。

■ do～end を使う

```
do
  繰り返したい処理
end
```

do ～ end で囲んだ部分が、ブロックになります。

二通りの書き方がありますが、場合によって自然に感じられる方を使用します。

なお、Rubyのブロックはメソッドに付加できるコードの塊であって,
それ自身は**オブジェクトではありません。**ブロックをオブジェクト化したものを**クロージャ**と
呼びます。

◇ ブロック付きメソッド
メソッドにブロックを渡して呼び出すには、メソッドの後ろにブロックをくっつけます。

```
オブジェクト.メソッド名(引数1, 引数2, 引数3,・・・,引数n)  { |ブロック変数|
  繰り返したい処理
}
```

```
オブジェクト.メソッド名(引数1, 引数2, 引数3,・・・,引数n)  do |ブロック変数|
  繰り返したい処理
end
```

・ 呼び出し時のブロック内の処理を有効にするためには、メソッドの定義時に
yield 文を記述します。

```
def func
  yield
  yield
end

func {puts "Hello,World!"
 puts "こんにちは¥n"
}
```

出力

```
Hello,World!
こんにちは
Hello,World!
こんにちは
```

定義の中でyieldが記述された回数だけ、呼び出し時に渡されたブロックが有効になっています。

呼び出し **ブロック**

```
func
```

```
{puts "Hello,World!"
puts "こんにちは¥n"
```

実行

```
def func
yield
yield
end
```

```
{puts "Hello,World!"
puts "こんにちは¥n"
```

```
{puts "Hello,World!"
puts "こんにちは¥n"
```

なお、ブロックを引数として渡すときは、&をつけます。

・ yieldには引数を渡すことも出来ます。yieldの受け取った引数は、
呼び出し時に渡されたブロックのブロック変数に渡されます。

```
def alph
  yield "A"
  yield "B"
  yield "C"
end
alph {|x| puts x }
```

出力

```
A
B
C
```

■ **ブロック変数**は、繰り返しごとに違う値を持ちます。
繰り返しごとに違う値を取り出すことが出来ます。

```
def foo(val)
 val.each{| i |
  yield i
}
end

pg = ["Ruby", "C++", "Java", "C"]

foo (pg){ | i |
  print(i , "言語","¥n")
}
```

出力

```
Ruby言語
C++言語
Java言語
C言語
```

・ ブロックを引数のように渡すことも出来ます。
そのためには、Procクラスによってブロックをオブジェクトとして生成します。

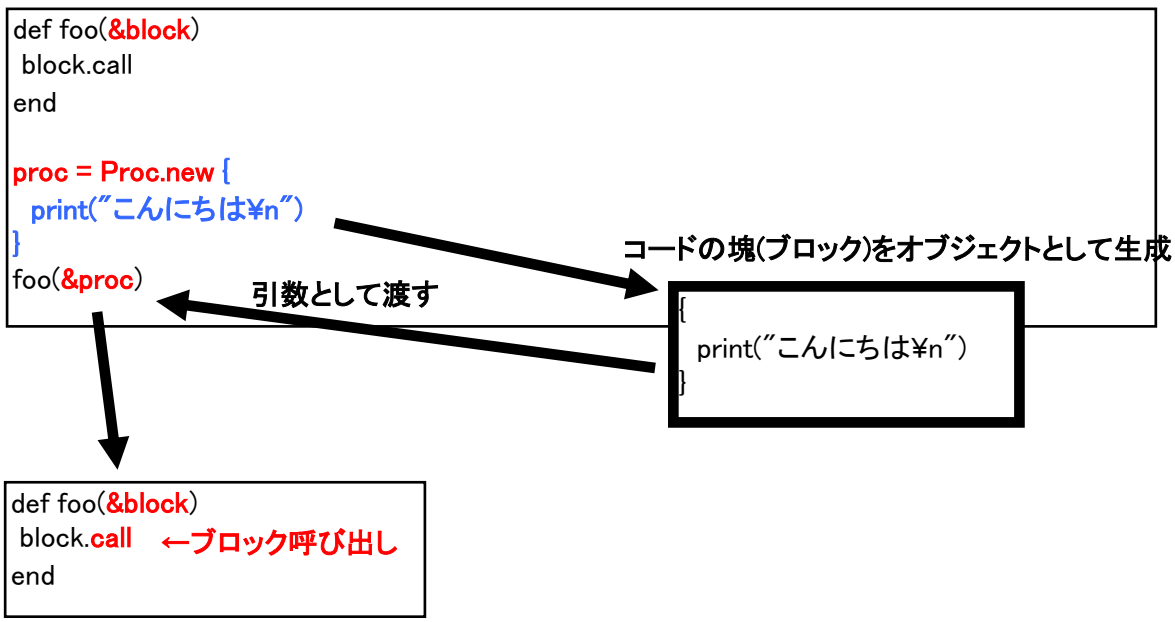
```
def foo(&block)
 block.call
end

proc = Proc.new {
  print("こんにちは¥n")
}
foo(&proc)
```

出力

```
こんにちは
```

このようなオブジェクトを、手続き型オブジェクトといいます。

渡されたブロックは、callメソッドで呼び出すことが出来ます。



● Fileクラス

・ ファイルを扱います。

◆ ファイルを開く

・ ファイルを開くには、open メソッドを使用します。
ファイルのオープンモードには、以下のものがあります。

"r"	読み込みモード
"r+"	読み書き両用モード
"w"	新規作成書き込みモード
"w+"	新規作成読み書き両用モード
"a"	追加書き込みモード
"a+"	追加読み書き両用モード
"rb","r+b","wb","w+b","ab","a+b"	上記6パターンのバイナリモード

◆ 出力

・ ファイルの内容を出力します。

eachメソッドを使う

```
f = File.open("filetest.txt")
if f
 puts "file open"
end

f.each {|line| print line}
f.close
```

getsメソッドを使う

```
f = File.open("filetest.txt")
if f
 puts "file open"
end

while line = f.gets
  print line
end
f.close
```

◆ ファイルへの書き込み

・ 読み込んだファイルへ書き込みます。

putsを使う

```
f = File.open("filetest.txt",'w')
if f
 puts "file open"
end

f.puts 'bar'

f.close
```

● デバッグについて

デバッグとは、プログラムの不具合（バグ）を見つけ出し、それを修正することを言います。

デバッグの作業の流れは大きく3つになります。
①発生しているバグの原因を見つける
②原因を取り除くためにプログラムを修正する
③修正が正しく行われているかどうか、再度プログラムを実行して確認する

その中でも特に難しいとされているのが、バグの原因を見つけることです。
プログラムの修正自体は簡単でも、そこに至るまでの原因を見つけることに時間を取られて
結果的にデバッグに割く時間が多くなってしまう、というケースもあります。

そうした問題を解決するために、デバッグについて様々な方法やツールが考案されてきました。
これらに関しては大勢の人々が議論し、多くの文献にも取り上げられています。

デバッグの方法は、大きく静的デバッグと動的デバッグの2つに分けられます。
それぞれの具体的な方法の例として、次のようなものがあります。

・静的デバッグ
机上デバッグを行う
コードレビューをする

・動的デバッグ
途中経過を出力する
デバッガを使う

静的デバッグについては本書で詳しく取り上げませんが、
様々な方法が公開されていますので、興味のある方はそちらを参考にしてみてください。

動的デバッグで一般的な方法は、途中経過を出力する方法とデバッガを用いる方法です。
本書ではその2つについて、具体的な例を上げて紹介します。

● 途中経過を出力する

プログラムを実行した時、基本的には結果だけが表示されるようになっています。
ですが、何らかのバグが発生した時に重要となってくるのは、そこに至るまでの途中経過です。
何故なら、その途中経過の中にバグの原因などが隠されている場合が多いからです。

デバッグをする時には、途中経過を画面に出力して確認する方法が非常に効果的です。
C言語などの多くの言語では、画面に出力する際にprintf関数というものを使う場合が多いので、
この方法は「printfデバッグ」と呼ばれていることもあります。
Rubyにもprintfメソッドはありますがよりデバッグの目的に適している出力系メソッドの、
pメソッドやputsメソッドを使用してデバッグを行うことが多いです。

以下に例を挙げて、この方法について詳しく説明します。

例1

```
# 配列の中から10以下の数字を合計して出力するプログラム

numbers = [1, 21, 77, 4, 19, 5, 10]  # 数字の配列
sum = 0                              # 合計

numbers.each do | num |
  if num < 10
    sum += num
  end
end

puts "合計は#{sum}です"              # 想定では 20 が出力される
```

例1を実行すると、出力結果は以下のようになります。

```
合計は10です
```

合計が「20」になるはずが、実際には「10」になってしまいました。
察しの良い方はもう既にお気付きになられたと思いますが、
もし一目で原因が分からなかった場合はその原因を特定しなくてはなりません。

次のページでは、実際に途中経過を画面に出力して原因を特定してみましょう。

では実際に、先ほどのプログラムを用いてデバッグを行ってみましょう。

現在は最終的な合計の結果だけが出力されていますので、putsメソッドを使って
計算の経過を画面に出力させることで、より原因を分かりやすくします。

例2

```
# 配列の中から10以下の数字を合計して出力するプログラム

numbers = [1, 21, 77, 4, 19, 5, 10]  # 数字の配列
sum = 0                              # 合計

numbers.each do | num |
 if num < 10
  puts num                           # 何が合計されているかを出力する
  sum += num
 end
end

puts "合計は#{sum}です"              # 想定では 20 が出力される
```

例2を実行すると、出力結果は以下のようになります。

```
1
4
5
合計は10です
```

出力結果を見ると、10が合計に含まれていないことが確認できます。
目的は「10以下」の数字を合計することですので、
10を含めることが出来ればこの問題は解決することが分かりました。

それを踏まえてif文の条件を見直してみると、「num ＜ 10」つまり「10より下」となっていました。
これが原因となっているようなので、次にこの部分を修正してみましょう。

これまでのことを踏まえて、例3のようにプログラムを修正しました。

例3

```
# 配列の中から10以下の数字を合計して出力するプログラム

numbers = [1, 21, 77, 4, 19, 5, 10]  # 数字の配列
sum = 0                              # 合計

numbers.each do | num |
 if num <= 10                        # 条件を変更する
  # puts num                         # 動作確認のためにコメントアウト
  sum += num
 end
end

puts "合計は#{sum}です"              # 想定では 20 が出力される
```

例3を実行すると、出力結果は以下のようになります。

```
合計は20です
```

出力結果から合計が20になったことが確認出来ます。
これで無事、当初の目的通りのプログラムが完成しました。

今回使用したputsメソッドだけでなく、pメソッドでは違った出力形式で確認が出来ますし、
Loggerというクラスを使用すれば、ログの中に出力して確認が出来るようになります。
上手くいかない場合などには効果的に働くことも多いので、是非活用してみてください。

● デバッガ

デバッガとは、その名の通りデバッグを行うためのツール・ライブラリのことです。
ここではライブラリとしてのデバッガを使用した例を紹介したいと思います。

Rubyにはデバッガとなるライブラリがいくつか存在しており、「ruby-debug」などが一般的です。
「ruby-debug」も強力なデバッガであり、是非とも活用してほしいものではありますが、
今回は操作性を重視して「pry」というライブラリを紹介します。

「pry」は厳密に言えばデバッガではなく、以前本書で紹介した「irb」に近いものです。
ただし「pry」は「ruby-debug」と似た機能を搭載することが可能で、
これを使用することでデバッガとしても高い性能を発揮することが出来ます。
更に加えて、前述したような「irb」のような操作性と、「pry」ならではの機能などが合わさり、
非常に便利なライブラリとなっています。

早速、この「pry」を使用してデバッグを行ってみましょう。

例4

```
# ユーザ情報を表示用に整形してから出力するプログラム

user_data = {
                "id": "testuser01",                 #
                "name": "テストユーザー",           #
                "age": "20",                        # キーと値が両方とも文字列の想定
                "email": "fugafuga@test.co.jp",     #
                "tel": "090-XXXX-XXXX"              #
            }

user_data.each do | key, val |
  puts key + " : " + val                            # 想定では文字列が連結されて順に表示される
end
```

例4を実行すると、出力結果は以下のようになります。

```
output_test.rb:12:in `block in <main>': undefined method `+' for :id:Symbol (NoMethodError)
                from output_test.rb:11:in `each'
                from output_test.rb:11:in `<main>'
```

プログラムには問題がないように見えるのに、実行結果がエラーとなってしまいました。
今回はこの原因を「pry」で特定し、修正するところまで一貫してやってみたいと思います。

「pry」を使用するための準備として、まずはインストールを行います。
コマンドラインツールなどで以下のコマンドを実行してください。

```
gem install pry
```

インストールが完了すれば、プログラムを例5のように修正します。

例5

```
# ユーザ情報を表示用に整形してから出力するプログラム

require 'pry'                                    # ライブラリを読み込む

user_data = {
                "id": "testuser01",              #
                "name": "テストユーザー",        #
                "age": "20",                     # キーと値が両方とも文字列の想定
                "email": "fugafuga@test.co.jp",  #
                "tel": "090-XXXX-XXXX"           #
            }

user_data.each do | key, val |
 binding.pry                                     # ここで一旦処理が中断され、pryが起動する
 puts key + " : " + val                          # 想定では文字列が連結されて順に表示される
end
```

次にこのプログラムを実行してみましょう。

例5を実行すると、出力結果は以下のようになります。

```
Frame number: 0/2

From: /home/takegawa/ruby_test/output_test2.rb @ line 15 :

   10:                     "tel": "090-XXXX-XXXX"     #
   11:                 }
   12:
   13: user_data.each do | key, val |
   14:   binding.pry                                  # ここで一旦処理が中断され、pryが起動する
 => 15:   puts key + " : " + val
   16: end

[1] pry(main)>
```

上記の出力結果を見ると、画面に結合した文字列が出力される前の段階で処理を中断し、
「pry」が起動しているのが確認出来ます。

前述した通り、この「pry」は「irb」と近いものなので、
この状態からRubyのコードを作成して実行することが出来ます。
ただし「irb」とは違い、中断した時点での変数やプログラムの実行結果が保存された状態で、
新たにコードを作成して実行することが出来ます。

つまり上記の例ならば、出力する直前のkeyやvalの変数に何が入っているのか
などが確認出来る上に、更にそれを使って以降のコードの修正方法を探ることも出来ます。

ここで先ほどエラーになった時のメッセージをもう一度見てください。
メッセージにはkeyの中身である「id」がシンボルだと表示されています。
まずは変数にそれぞれ何が入っているのか確認してみましょう。

```
[1] pry(main)> key
=> :id
[2] pry(main)> val
=> "testuser01"
```

出力結果を見ると、ハッシュの値(val)は想定通り文字列が格納されていましたが、
同じ文字列で想定していたキー(key)がシンボルになっています。
これはRuby1.9以降のハッシュの記法を使用したことによるもので、
文字列で指定したつもりがシンボルに変換されていたようです。※1
エラーの原因は、このシンボルと文字列をそのまま連結させようとしていたことに
あったようなので、この部分を修正したいと思います。

ただし最初から元のプログラムを修正するのではなく、
修正方法を試してみて実際に期待通りの結果が得られることを先に確認してみましょう。

※1 Ruby2.2以降のバージョンでは値に違いがあるなど、場合によって挙動が変化します。

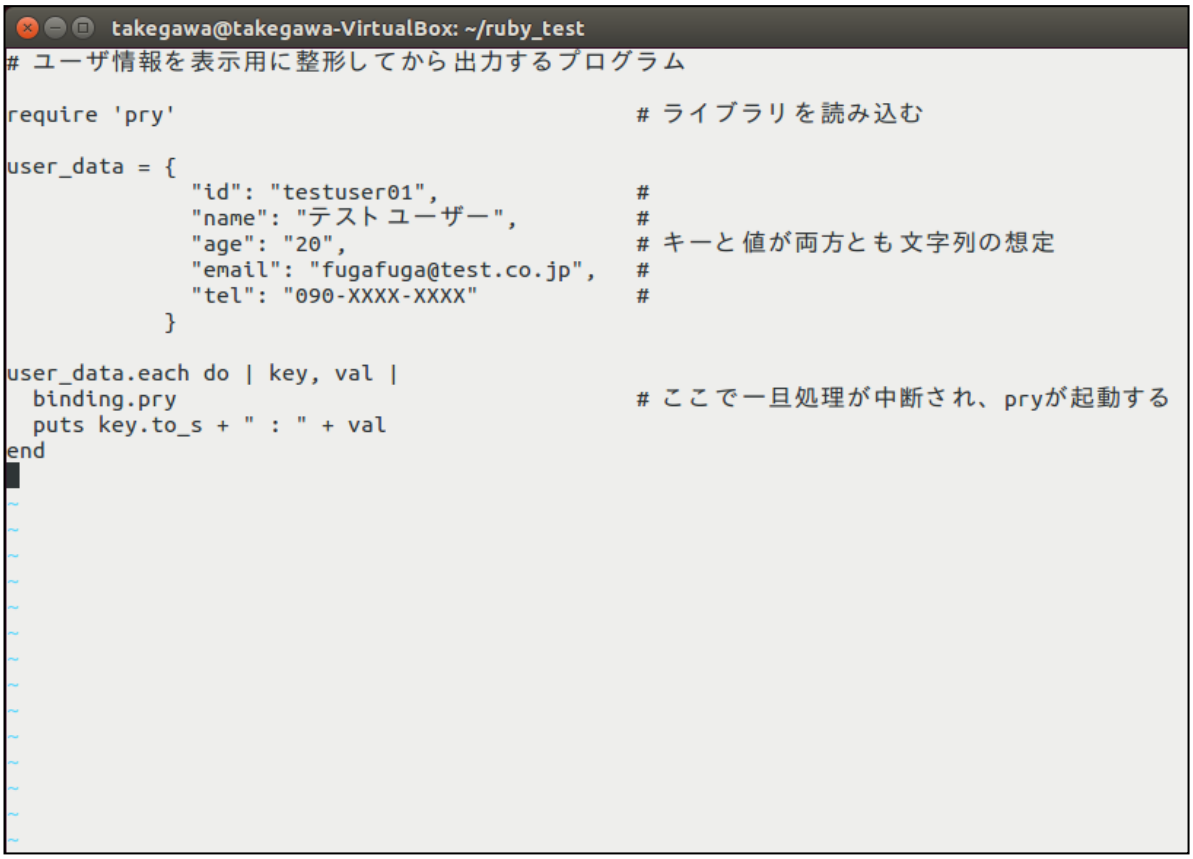
```
[3] pry(main)> key.to_s
=> "id"
[4] pry(main)> puts key.to_s + " : " + val
id : testuser01
=> nil
```

まずはシンボルを文字列に変換出来ることを、to_sメソッドを使って確認します。
そして変換した後に連結すれば問題無く出力出来ることも確認します。

これで修正方法が上手くいきそうだという見通しが立ったので、
実際に元のプログラムを修正してみたいと思います。
せっかくなので、その修正自体もこの「pry」からやってみましょう。

```
[5] pry(main)> edit [file_name]
```

editコマンドで[file_name]の部分に自分で作成したプログラムのファイル名を入れると
デフォルトに設定されているエディタ(vi、vim、nanoなど)が起動します。



起動したエディタでプログラムを例6のように修正します。

例6

```
# ユーザ情報を表示用に整形してから出力するプログラム

require 'pry'                                    # ライブラリを読み込む

user_data = {
                id: "testuser01",                #
                name: "テストユーザー",          #
                age: "20",                       # 不要な " を削除してコードを整理
                email: "fugafuga@test.co.jp",    #
                tel: "090-XXXX-XXXX"             #
            }

user_data.each do | key, val |
 # binding.pry                                   # 動作確認のためにコメントアウト
 puts key.to_s + " : " + val                     # シンボルを文字列に変換するように修正
end
```

保存してエディタを終了させると、元の「pry」に戻ってきます。
そこでexit!コマンドを使用して、一旦今の実行状態から抜け出します。

```
[6] pry(main)> exit!
```

先ほど例6のように修正したプログラムを再度実行すると、出力結果は以下のようになります。

```
id : testuser01
name : テストユーザー
age : 20
email : fugafuga@test.co.jp
tel : 090-XXXX-XXXX
```

無事全てのデータが整形されて表示出来るようになりました。
これでこのプログラムのデバッグは完了です。

● その他のコマンド

これまでに紹介したコマンドの他にも、「pry」には多くのコマンドが実装されています。
その中でも便利なものをいくつか紹介します。

・show-method [オブジェクト名]
オブジェクト名を指定すると、定義されているメソッドを検索して表示してくれるコマンドです。
特に誰かと共同開発をしている場合に、自分以外の人が作成したメソッドを
把握したい時などには非常に有用なコマンドです。

・find-method [メソッド名]
メソッド名を指定すると、該当するメソッドを検索して表示してくれるコマンドです。
単純にメソッド名で検索したい場合などには、このコマンドが効果的です。

・cd, ls
コマンドプロンプトでの同名コマンドと似たような機能を持ちます。
それぞれがpryの構造に対応しており、オブジェクト間を移動したり、
現在選択しているオブジェクトのメソッドや変数などを一覧で表示してくれたりします。

・shell-mode
シェルのコマンド入力が可能になる状態へ移行するコマンドです。
ただシェルコマンドが使えるだけでなく、コマンド内で#{ }の式展開が使用できます。

・help
「pry」で使用できるコマンドを一覧で確認することが出来るコマンドです。
各コマンドの詳細に関しては、コマンド名のオプションに -h を付けることで確認出来ます。

● プラグイン

「pry」には様々な面で活躍するプラグインも多く提供されています。
こちらも一部紹介したいと思います。

・pry-debug
よりデバッガとしての機能を充実させるためのプラグインです。
一行ずつ実行したりなどの細かいデバッグ作業が可能になります。

・pry-doc
「pry」からドキュメントを参照できるようになるプラグインです。
Rubyの内部実装をC言語表示することも出来るようになるので、
C言語を学んだことがある人や、より詳しく掘り下げたい人などにもオススメです。

・pry-rails
Ruby on Rails を使用して開発する場合に効果的なプラグインです。
Railsアプリの再起動や、modelやroutesの検索が容易になったりと、
Railsアプリ開発の助けとなる機能が充実しています。

今回は全体を通して「pry」について紹介をしましたが、
最初に紹介した「ruby-debug」など、Rubyで使用できるデバッガはいくつか存在します。
是非可能な限り多くに触れてみて、自身の手に合ったものを活用してみてください。

2016年8月1日　第1刷

2021年3月24日　第2刷