Ruby 実習マニュアル

第五版

大黒学

Ruby 実習マニュアル・第五版
著者——大黒学

2001 年 11 月 12 日（月）　第零版発行
2004 年　1 月　5 日（月）　第一版発行
2004 年　3 月　8 日（月）　第二版発行
2005 年　3 月　6 日（日）　第三版発行
2006 年　3 月　2 日（木）　第四版発行
2007 年　8 月 19 日（日）　第五版発行

目次

第 1 章　オブジェクト指向プログラミングの基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したもののことを、「文書」(document) と呼びます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどういう意味になるかということを定めた規則が必要になります。そのような規則は、「言語」(language) と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」(natural language) と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」(artificial language) と呼ばれるものもあります。人間ではなくコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、普通、自然言語ではなく人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」(program) と呼ばれます。

プログラムを作成するためには、ただ単にそれを書くという作業だけではなく、その構造を設計したり、その動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」(programming) と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語には、たくさんのものがあります。たとえば、Pascal、C、Lisp、ML、Prolog、Smalltalk、Eiffel、……というように、枚挙にいとまがないほどです。それぞれのプログラミング言語は、自分にとって得意なことと不得意なことを持っていて、万能のプログラミング言語というものは存在しません。ですから、プログラムを書くときは、その目的に応じて適切なプログラミング言語を選択することが重要です。

1.1.4　スクリプト言語

プログラミング言語を設計するときには、どのようなことを得意とする言語を作るのかという方針を立てる必要があります。複数の方針を立ててプログラミング言語を設計することも可能です。しかし、プログラムがコンピュータによって実行されるときの効率を向上させるという方針と、プログラムが人間にとって書きやすくて読みやすいものになるようにするという方針とは、トレードオフの関係にあります。つまり、それらの二つの方針を両立させることは、とても困難なことなのです。

プログラミング言語の中には、書きやすさや読みやすさよりも実行の効率を優先させて設計されたものもあれば、それとは逆に、効率よりも書きやすさや読みやすさを優先させて設計されたものもあります。後者の方針で設計された言語は、「スクリプト言語」(scripting language) と呼ばれます。スクリプト言語としては、sed、awk、Perl、Tcl、Python、Ruby などがよく使われています。

プログラミング言語で書かれた文書は「プログラム」と呼ばれるわけですが、スクリプト言語で書かれた文書は、「スクリプト」(script) と呼ばれることが多いようです。

1.1.5　言語処理系

コンピュータというものは異質な二つの要素から構成されていて、それぞれの要素は、「ハードウェア」(hardware) と「ソフトウェア」(software) と呼ばれます。ハードウェアというのは物理的な装置のことで、ソフトウェアというのはプログラムなどのデータのことです。

コンピュータは、さまざまなプログラミング言語で書かれたプログラムを理解して実行することができます。しかし、コンピュータのハードウェアが、ソフトウェアの助力を得ないで単独で理解することのできるプログラミング言語は、ハードウェアの種類によって決まっているひとつの言語だけです。

ハードウェアが理解することのできるプログラミング言語は、そのハードウェアの「機械語」(machine language) と呼ばれます。機械語というのは、人間にとっては書くことも読むことも困難な言語ですので、人間が機械語でプログラムを書くということはめったにありません。

人間にとって書いたり読んだりすることが容易なプログラミング言語で書かれたプログラムをコンピュータに理解させるためには、そのためのプログラムが必要になります。そのような、人間が書いたプログラムをコンピュータに理解させるためのプログラムのことを、「言語処理系」(language processor) と呼びます（「言語」を省略して、単に「処理系」と呼ぶこともあります)。

言語処理系には、「コンパイラ」(compiler) と「インタプリタ」(interpreter) と呼ばれる二つの種類があります。コンパイラというのは、人間が書いたプログラムを機械語に翻訳するプログラムのことで、インタプリタというのは、人間が書いたプログラムがあらわしている動作をコンピュータに実行させるプログラムのことです。

1.2　オブジェクト

1.2.1　プログラミングパラダイム

人間がプログラムを書くためには、そのプログラムによって実現したいことを何らかのプログラミング言語を使って具体的に記述する必要があります。しかし、「実現したいこと」と「具体的な記述」とのあいだには、大きな距離があります。プログラムを書く人間は、思考することによって、その距離を埋める必要があります。

どのように考えれば、「実現したいこと」と「具体的な記述」とのあいだにある距離を埋めることができるか、という考え方のことを、「プログラミングパラダイム」(programming paradigm) と呼びます。

プログラミングパラダイムは、厳密に言えば、プログラミング言語ごとに異なるわけですが、それらは、いくつかの類型に大きく分類することが可能です。プログラミングパラダイムの類型としては、「オブジェクト指向プログラミング」(object-oriented programming)、「関数プログラミング」(functional programming)、「論理プログラミング」(logic programming)、「手続き型プログラミング」(procedural programming) などがあります。

ところで、この「Ruby 実習マニュアル」という文章は、Ruby というプログラミング言語を使ってプログラムを書く方法について解説することを目的として書かれたものです。

Ruby を使ってプログラムを書く場合に使用されるプログラミングパラダイムは、オブジェクト指向プログラミングです。そこで、この節では、オブジェクト指向プログラミングにおいて中心的な役割を果たす、「オブジェクト」と呼ばれるものについて説明したいと思います。

1.2.2　オブジェクトとは何か

「オブジェクト」(object) というのは、箱のようなものだと考えることができます。オブジェクト指向プログラミングにおいては、いくつかのオブジェクトに対する操作という形で、プログラムを記述していきます。

オブジェクトという箱の中には、2 種類のものが詰め込まれています。ひとつの種類はデータで、もうひとつの種類は、「メソッド」(method) と呼ばれるものです。

メソッドというのは、何らかの仕事をするもののことです。メソッドの仕事というのは、基本的には、自分が所属しているオブジェクトの中にあるデータの操作です。ひとつのオブジェクトはいくつかのメソッドを持っていて、それぞれのメソッドは、自分に固有の仕事を実行します。ですから、オブジェクトというのは、自分の中にあるデータを操作するためのさまざまな機能を持っている箱のことだと考えることができます。

オブジェクトにはさまざまな種類があります。たとえば文字列のオブジェクトや数値のオブジェクトなどです。オブジェクトがどんなメソッドを持っているかというのは、そのオブジェクトの種類によって決まっています。たとえば、文字列のオブジェクトは、文字列から文字を取り出すメソッドや、文字の個数を数えるメソッドや、大文字を小文字に変換するメソッドなどを持っています。

1.2.3 メッセージ

メソッドに仕事をさせることができるのは、そのメソッドを持っているオブジェクトだけです。

オブジェクトに対してメソッドに仕事をさせる依頼をするためには、「メッセージ」と呼ばれるものをそのオブジェクトに送る必要があります[1]。「メッセージ」(message) というのは、「このメソッドにこんな仕事をさせてください」という意味を持つ記述のことです。

オブジェクトは、メッセージを受け取ると、その記述にしたがって、自分の中にある適切なメソッドに仕事をさせます。

オブジェクトにメッセージを送ることによってメソッドに仕事をさせることを、メソッドを「呼び出す」(call) と言います。また、メッセージを受け取ったオブジェクトは、そのメッセージの「レシーバー」(receiver) と呼ばれます。

オブジェクト指向プログラミングというのは、「オブジェクトにメッセージを送る」ということを基本とするプログラミングパラダイムのことです。

1.2.4 引数と戻り値

メソッドは、自分が呼び出されたとき、自分の仕事を開始するのに先立って、自分を呼び出したものから何個かのオブジェクトを受け取ることができます。メソッドが受け取るオブジェクトのそれぞれは、「引数」(argument) と呼ばれます。

また、メソッドは、自分の動作が終了したのち、かならず、自分を呼び出したものに対して 1 個のオブジェクトを返します。メソッドが返すオブジェクトは、「戻り値」(returned value) と呼ばれます。

1.3 Ruby の概要

1.3.1 Ruby とは何か

前の節にも書きましたが、この「Ruby 実習マニュアル」という文章は、Ruby というプログラミング言語を使ってプログラムを書く方法について解説することを目的として書かれたものです。そこで、Ruby というプログラミング言語についての本格的な説明に入る前に、この節で、それについて概略的な紹介をしておきたいと思います。

Ruby というのは、まつもとゆきひろさんという人によって設計されたプログラミング言語です。この言語が持っている特徴はいくつもあるのですが、それらのうちでとりわけ重要なのは、

- スクリプト言語である。
- オブジェクト指向プログラミングに基づいて設計されている。

という二つの特徴です。

1.3.2 式と評価と値

Ruby のプログラムは、「式」(expression) と呼ばれるものから構成されます。式というのは、何らかの動作を記述した文字の列のことです。式が記述している動作を実行することを、その式を「評価する」(evaluate) と言います。式を評価すると、その結果として 1 個のオブジェクトが得られます。式を評価することによって得られたオブジェクトのことを、その式の「値」(value) と言います。

1.3.3 リテラル

特定のオブジェクトを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。リテラルを評価すると、それによって生成されたオブジェクトが、そのリテラルの値になります。

リテラルにはさまざまな種類があるのですが、ここでは、整数と文字列のオブジェクトを生成するリテラルについて、ごく簡単に説明しておきたいと思います[2]。

0 から 9 までの数字を並べてできる列は、整数のオブジェクトを生成するリテラルです。たとえば、2800 というのは、整数のオブジェクトを生成するリテラルの一例です。このリテラルを評価すると、2800 という整数のオブジェクトが生成されて、そのオブジェクトが値として得られます。

---

[1] オブジェクトにメッセージを送る方法については、第 1.4 節で説明します。

[2] リテラルについては、第 2.2 節で、さらに詳細に説明します。

文字列のオブジェクトを生成するリテラルは、その文字列を二重引用符 (") で囲んだものです。たとえば、"namako" というのは、文字列のオブジェクトを生成するリテラルの一例です。このリテラルを評価すると、namako という文字列のオブジェクトが生成されて、そのオブジェクトが値として得られます。

1.3.4　Ruby の処理系

Ruby の処理系としては、現在のところ、ruby という名前[3]のインタプリタが、もっとも普及しています。このインタプリタは、Linux ディストリビューションの大多数と MacOS には最初から組み込まれていますので、それらの OS を使っている場合、インストールは不要です。

Ruby のプログラムを実行したいけれども、使っている OS に Ruby の処理系が組み込まれていない、という場合は、処理系をインストールする必要があります。ruby は、無料で配布されていて、インターネット上のサイトからダウンロードすることができます。ruby をダウンロードしたりインストールしたりする方法について知りたい方は、

Ruby 公式サイト　http://www.ruby-lang.org/ja/

を参照してください。

1.3.5　対話型のインタプリタ

人間が入力したものを読み込んでそれを処理する、という動作を延々と繰り返すように作られているプログラムは、「対話型である」(interactive) と言われます。

インタプリタには、対話型のものとそうでないものとがあります。ruby というのは対話型のインタプリタではないのですが、irb というプログラムを使うことによって、ruby を対話型のインタプリタとして使うことができるようになります。

それでは、実際に irb を起動してみましょう。まず、シェル（Linux ならば bash など、Windows ならば cmd.exe など）を起動して、そのシェルに対して irb というコマンドを入力してみてください。すると、irb が起動して、

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

というプロンプトが表示されるはずです。

irb は、

(1) プロンプトを出力する。
(2) 式を読み込む。
(3) 読み込んだ式を評価する。
(4) 評価によって得られた値を出力する。

ということを延々と繰り返すように作られています。

それでは、整数のオブジェクトを生成するリテラルを irb に入力してみましょう。たとえば、

```
irb(main):001:0> 2800
```

というように、整数のリテラルを入力して、そののちエンターキーを押してみてください。すると、

```
=> 2800
```

と出力されます。=> というのは、その右側に出力されているものが式の値だということを示している矢印です。リテラルを評価すると、それによって生成されたオブジェクトが値として得られますので、2800 というリテラルを評価した結果が、矢印の右側に 2800 と出力されたわけです。

次に、文字列のオブジェクトを生成するリテラルを irb に入力してみましょう。そうすると、

```
irb(main):002:0> "namako"
=> "namako"
```

というように、文字列のリテラルを評価することによって得られた文字列のオブジェクトが、矢印の右側に出力されます。

それでは次に、irb を終了させてみましょう。irb は、exit という式を入力することによって終了させることができます。それでは、実際に exit を入力して、irb を終了させてみてください。

[3]先頭の文字が大文字になっている Ruby は言語の名前で、小文字になっている ruby はインタプリタの名前です。

1.4 メッセージ

1.4.1 メッセージについての復習

この節では、メッセージについて説明したいと思います。

メッセージとは何か、ということについては、すでに第 1.2 節で説明していますので、まず、そこで説明したことを復習しておくことにしましょう。

「メッセージ」(message) というのは、オブジェクトに対して送られる、「このメソッドにこんな仕事をさせてください」という意味を持つ記述のことです。オブジェクトは、メッセージを受け取ると、その記述にしたがって、自分の中にある適切なメソッドに仕事をさせます。

オブジェクトにメッセージを送ることによってメソッドに仕事をさせることを、メソッドを「呼び出す」(call) と言います。また、メッセージを受け取ったオブジェクトのことを、そのメッセージの「レシーバー」(receiver) と呼びます。

1.4.2 メッセージ式

第 1.2 節では、オブジェクトにメッセージを送るためにはどうすればいいかということについては、まったく説明していませんでした。それではさっそく、その方法について説明しましょう。

オブジェクトにメッセージを送りたいときは、「メッセージ式」(message expression) と呼ばれる式を書きます。

メッセージ式というのは、

式 . メッセージ

という形の式のことです。メッセージ式の中にあるドット (.) は、「左側の式の値に右側のメッセージを送る」という意味を持っています。

メッセージ式を評価すると、まず、ドットの左側に書かれた式が評価されます。そして、その値として得られたオブジェクトに対して、ドットの右側の部分がメッセージとして送られます。つまり、ドットの左側に書かれた式の値をレシーバーにして、メッセージが送られるわけです。

メッセージ式を評価することによって得られる値は、それによって呼び出されたメソッドが返した戻り値です。

1.4.3 引数を渡さないメッセージ

次に、メッセージの書き方について説明しましょう。ただし、引数を受け取るメソッドを呼び出すためのメッセージの書き方はちょっと複雑なので、それは後回しにして、まずは、引数を受け取らないメソッドを呼び出すためのメッセージの書き方について説明します。

実は、引数を受け取らないメソッドを呼び出すためのメッセージというのは、そのメソッドの名前をそのまま書くだけです。

それでは、具体的な例を使って説明しましょう。文字列のオブジェクトは、 `downcase` というメソッドを持っています。これは、レシーバーに含まれているすべての英大文字を英小文字に変換することによってできる文字列のオブジェクトを戻り値として返す、という動作をするメソッドです。たとえば、 `hItODe` という文字列のオブジェクトが持っている `downcase` を呼び出すと、その戻り値として、 `hitode` という文字列のオブジェクトが得られます。

`downcase` は引数を受け取らないメソッドですから、 `downcase` という名前が、そのまま、そのメソッドを呼び出すためのメッセージになります。つまり、文字列のオブジェクトに `downcase` というメッセージを送ると、そのオブジェクトは、自分の中にある `downcase` というメソッドに仕事をさせることになる、ということです。

オブジェクトにメッセージを送りたいときは、

式 . メッセージ

というメッセージ式を書けばいいわけですから、たとえば、

```
"hItODe".downcase
```

というメッセージ式を書くことによって、 `hItODe` という文字列に対して `downcase` というメッセージを送ることができます。

メッセージ式を評価することによって得られる値は、それによって呼び出されたメソッドが返した戻り値ですので、

```
"hItODe".downcase
```

というメッセージ式を評価すると、それによって呼び出された downcase が返した戻り値、つまり hitode という文字列のオブジェクトが、そのメッセージ式の値になります。

それでは、irb を使って、文字列のオブジェクトに downcase というメッセージを送ってみましょう。そうすると、次のようになります。

```
irb(main):001:0> "hItODe".downcase
=> "hitode"
```

1.4.4　引数を渡すメッセージ

次に、引数を受け取るメソッドを呼び出すためのメッセージの書き方について説明しましょう。

まず、説明のために使うメソッドの実例として、 gsub というメソッドを紹介します。これは、文字列のオブジェクトが持っているメソッドで、部分文字列の置き換え (substitution) という動作を実行します。

「部分文字列」(substring) というのは、文字列の中に含まれている文字列のことです。たとえば、 namako という文字列は、 nama 、 ama 、 am 、 m などの部分文字列を含んでいます。

gsub は、2 個の引数を受け取ります。1 個目は探索する文字列のオブジェクトで、2 個目は、発見された部分文字列を置き換える文字列のオブジェクトです。 gsub は、レシーバーの中で、1 個目の引数と一致する部分文字列を探索して、発見されたすべての部分文字列を 2 個目の引数に置き換えることによってできる文字列のオブジェクトを戻り値として返す、という動作をします。たとえば、レシーバーが、

```
ccxyzcccxyzcxyzcccxyzc
```

という文字列のオブジェクトで、1 個目の引数が xyz で、2 個目の引数が 789 だとすると、 gsub は、

```
cc789ccc789c789ccc789c
```

という文字列のオブジェクトを戻り値として返します。

gsub のような、引数を受け取るメソッドを呼び出すためには、どのような引数をメソッドに渡すのかということを記述したメッセージをオブジェクトに送る必要があります。そのようなメッセージは、

メソッド名 (式 , ···)

と書きます。この形のメッセージをオブジェクトに送ると、丸括弧の中に書かれた式の値が、書かれた順番のとおりに、引数としてメソッドに渡されることになります。ですから、 gsub を呼び出したいときは、

gsub(式 $_1$, 式 $_2$)

という形のメッセージを文字列のオブジェクトに送ればいいわけです。そうすると、式 $_1$の値が 1 個目の引数として gsub に渡されて、式 $_2$の値が 2 個目の引数として gsub に渡されます。

それでは、irb を使って、実際に試してみましょう。

```
irb(main):001:0> "ccxyzcccxyzcxyzcccxyzc".gsub("xyz", "789")
=> "cc789ccc789c789ccc789c"
```

ちなみに、 gsub というメソッドの名前は、globally substitute（大域的に置き換える）という言葉を縮めて作られたものです。「大域的に」というのは、「文字列全体に渡って」という意味です。 gsub とは別に、大域的ではない置き換えを実行する sub というメソッドもあります。 sub は、レシーバーの先頭から末尾に向かって部分文字列を探索して、最初に発見された部分文字列だけを置き換えます。irb を使って試してみると、次のようになります。

```
irb(main):001:0> "ccxyzcccxyzcxyzcccxyzc".sub("xyz", "789")
=> "cc789cccxyzcxyzcccxyzc"
```

1.4.5　メソッドの戻り値にメッセージを送る式

メソッドの戻り値に対して、さらにメッセージを送りたい、ということがしばしばあります。

そんなときは、メッセージ式のうしろに、さらにドットとメッセージを書きます。つまり、

式 . メッセージ $_1$. メッセージ $_2$

という形の式を書けばいいわけです。この形の式は、まず左端にある式の値に対してメッセージ$_1$を送って、その結果として得られたオブジェクトに対してさらにメッセージ$_2$を送る、という意味だと解釈されます。

たとえば、

```
"kITSutSUki".downcase.gsub("tsu", "mo")
```

という式を評価すると、まず kITSutSUki という文字列のオブジェクトに downcase というメッセージが送られます。すると、その結果として、 kitsutsuki という文字列のオブジェクトが得られます。そしてそののち、

```
gsub("tsu", "mo")
```

というメッセージが、 kitsutsuki という文字列のオブジェクトに送られます。ですから、式全体の値として、 kimomoki という文字列のオブジェクトが得られることになります。

1.4.6 トップレベル

メッセージ式は、普通、レシーバーを求める式とドットを書いて、その右側にメッセージを書くわけですが、場合によっては、レシーバーを求める式とドットを省略することが可能です。レシーバーを省略した場合、メッセージは、文脈によって決定されるオブジェクトに送られます。

irb に入力した式は、「トップレベル」(toplevel) と呼ばれる文脈で評価されます。レシーバーを省略したメッセージ式をトップレベルで評価すると、そのメッセージは、「トップレベルオブジェクト」(toplevel object) と呼ばれるオブジェクトに送られます。

irb を終了させたいときに入力する exit という式は、レシーバーが省略されたメッセージ式です。ですから、irb に exit という式を入力すると、トップレベルオブジェクトに対して exit というメッセージが送られるわけです。その結果、トップレベルオブジェクトの中にある exit というメソッドが呼び出されて、そのメソッドが irb を終了させます。

1.4.7 関数的メソッド

メソッドというのは、基本的にはレシーバーに対して何らかの処理を実行するわけですが、例外的に、レシーバーとは無関係な動作をするメソッドもあります。そのような、レシーバーに対する処理を何もしないメソッドは、「関数的メソッド」(functional method) あるいは単に「関数」(function) と呼ばれます。たとえば、 exit というメソッドは、関数的メソッドのひとつです。関数的メソッドは、たいていの場合、レシーバーを省略したメッセージ式を書くことによって呼び出すことができます。

Ruby の処理系にはさまざまな関数的メソッドが組み込まれているのですが、ここでは、その例として system というメソッドを紹介しておきたいと思います。

system というのは、引数として文字列を受け取って、それをシェルのコマンドとして実行する、という動作をする関数的メソッドです。

もしも、使っている OS が Linux ならば、

```
system("cal 8 1997")
```

という式を irb に入力してみてください。そうすると、1997 年 8 月のカレンダーが出力されるはずです。Linux ではなくて Windows を使っているならば、

```
system("date /t")
```

という式を irb に入力してみましょう。そうすると、今日の日付が出力されるはずです。

system は、 true または false というオブジェクトを戻り値として返します。コマンドの実行に成功した場合は true で、失敗した場合は false です。

1.5 クラス

1.5.1 クラスとは何か

オブジェクトは、「クラス」(class) と呼ばれるものから生成されます。

クラスというのは、鋳型のようなものだと考えるといいでしょう。つまり、何らかの形を持っている容器だということです。液体をその容器に入れて、そののち何らかの方法でその液体を個体に変えて、そして中身を取り出すと、容器の中の空間と同じ形を持つ物体が出てきます。鋳型

を使わずに物体を作ることも可能ですが、鋳型を使うことによって、同じ形を持つ物体を何個も作るということが簡単にできるようになります。

オブジェクトというのは、クラスという鋳型から作られた物体のことだと考えることができます。ですから、ひとつのクラスから生成されたすべてのオブジェクトは、同じメソッドを持つことになります。

オブジェクトには、整数や文字列など、さまざまな種類があります。そして、オブジェクトを生成するクラスも、オブジェクトの種類ごとに別々になっています。つまり、整数は整数のクラスから生成され、文字列は文字列のクラスから生成される、ということです。

それぞれのクラスは、それを識別するための名前を持っています。たとえば、整数のオブジェクトを生成するクラスは、Fixnum という名前を持っています。同じように、文字列のオブジェクトは String という名前のクラスから生成されます。

1.5.2　インスタンス

「インスタンス」(instance) という言葉が、オブジェクト指向プログラミングに関連する文章や会話の中で、しばしば使われることがあります。

実は、「インスタンス」という言葉の意味は、基本的には「オブジェクト」とほとんど同じです。相違点は、「オブジェクト」よりも「インスタンス」のほうが、何らかのクラスから生成されたものだという観点が強調されるというところにあります。ですから、「これは String クラスのオブジェクトだ」という言い方は、決して間違いではありませんが、この場合は、「これは String クラスのインスタンスだ」という言い方のほうが、意味が鮮明です。

オブジェクトが何というクラスのインスタンスなのかということを調べたいときは、class というメソッドを使います。class は、すべてのオブジェクトが持っているメソッドで、レシーバーを生成したクラスを戻り値として返す、という動作をします。たとえば、

```
38.class
```

というように、整数のオブジェクトに class というメッセージを送る式を irb に入力すると、Fixnum というクラス名が出力されます（class メソッドはクラスそのものを返しているのですが、irb が出力するのはその名前だけです）。

それでは、文字列のオブジェクトについても、class を使って、それを生成したクラスの名前を確かめてみましょう。たとえば、

```
"hitode".class
```

という式を irb に入力すると、String というクラス名が出力されるはずです。

1.5.3　クラスを生成するクラス

実は、クラスというのもオブジェクトの一種です。ということは、クラスに対して class というメッセージを送れば、それを生成したクラスがわかるはずです。たとえば、irb に、

```
Fixnum.class
```

という式を入力してみてください。そうすると、Fixnum というクラスは Class というクラスのインスタンスだということがわかります。

Fixnum だけではなくて、すべてのクラスは、Class というクラスのインスタンスです。ちなみに、Class クラス自身も、やはり Class クラスのインスタンスです。つまり、Class クラスは、自分自身から生成されるわけです。

1.5.4　スーパークラスとサブクラス

すでに説明したように、クラスというのはオブジェクトを生成する鋳型のようなものですが、それはクラスというものの側面のひとつにすぎません。クラスには、もうひとつの別の側面があるのです。それは、「分類体系の部品」という側面です。

さまざまなクラスの全体は、生物などの分類体系と同じように、木の形をした構造を持っています。この木は、本物の木とは上と下とが逆になっていて、いちばん上に根があって、下に向かって枝が伸びています。木を構成するそれぞれのクラスは、上にあるものほど一般的で、下にあるものほど特殊です。

B というクラスが、A というクラスを特殊化することによって作られたものだとするとき、A のことを、B の「スーパークラス」(superclass) と言い、B のことを、A の「サブクラス」(subclass) と言います。

メッセージ式	説明
o.to_s	*o* をあらわす文字列を返します。
o.display	*o* をあらわす文字列を出力します。
o.class	*o* を生成したクラスを返します。

表 1.1: すべてのオブジェクトが持っている主要なメソッド

クラスのスーパークラスが何なのかということを調べたいときは、superclass というメソッドを使います。superclass は、すべてのクラスが持っているメソッドで、レシーバーのスーパークラスを戻り値として返します。

それでは、Fixnum クラスのスーパークラスが何なのかということを調べてみましょう。irb に、

```
Fixnum.superclass
```

という式を入力してみてください。すると、Integer という名前が出力されるはずです。つまり、Fixnum のスーパークラスは Integer というクラスだということです。

Integer のスーパークラスは、Numeric というクラスです。そして、Numeric と String のスーパークラスは、ともに Object というクラスです。

1.5.5 クラスの木の根

Ruby では、クラスの木を上へ上へとたどっていくと、かならず Object というクラスに到達します。つまり、Object というのが、クラスの木の根に相当するわけです。クラスの木の根ということは、もっとも一般的なクラスということです。

ほとんどすべてのクラスは、自分のスーパークラスというものを持っているわけですが、Object だけはその例外です。しかし、Object も、クラスである以上、superclass メソッドを持っているという点は、ほかのクラスと同じです。

それでは、irb を使って、superclass というメッセージを Object に送ってみましょう。そうすると、Object クラスが持っている superclass メソッドは、戻り値として nil というオブジェクトを返す、ということが分かります。ちなみに、nil というのは、「存在しない」ということなどをあらわすために使われるオブジェクトで、この場合は、「スーパークラスは存在しない」ということを意味しています。

1.5.6 組み込みクラス

Ruby の処理系の中には、さまざまなクラスが最初から組み込まれています。そのような、処理系の中に最初から組み込まれているクラスは、「組み込みクラス」(built-in class) と呼ばれます。これまでの説明の中で登場した、Fixnum、String、Class、Integer、Object などは、すべて組み込みクラスです。

プログラムが扱うことのできるクラスは、組み込みクラスだけではありません。プログラムを書く人は、クラスを作る記述[4]をプログラムの中に書くことによって、自分が必要とするクラスを自由に作り出すことができます。

1.6 主要なメソッド

1.6.1 すべてのオブジェクトが持っているメソッド

この節では、組み込みクラスのオブジェクトが持っているメソッドのうちの主要なものをいくつか紹介したいと思います。まず最初は、すべてのオブジェクトが持っているメソッドです。

表 1.1 は、すべてのオブジェクトが持っているメソッドのうちの主要なものを示しています。

何らかのクラスのオブジェクトを文字列のオブジェクトに変換したいときは、to_s というメソッドを使います。たとえば、

```
7803.to_s
```

[4] クラスを作る記述の書き方については、第 3 章で説明します。

メッセージ式	説明
s.size	s の長さ（文字数）を返します。
s.length	s.size と同じ意味です。s の長さを返します。
s.chomp	s の末尾にある改行を削除した結果を返します。
s.downcase	s に含まれる英大文字を英小文字に変換した結果を返します。
s.upcase	s に含まれる英小文字を英大文字に変換した結果を返します。
s.ljust(n)	s の右側を空白で埋めた、長さが n の文字列を返します。
s.rjust(n)	s の左側を空白で埋めた、長さが n の文字列を返します。
s.center(n)	s の左右を空白で埋めた、長さが n の文字列を返します。
s.delete(t)	t に含まれる文字を s から削除した結果を返します。
s.sub(t, u)	s に含まれる最初の t を u に置き換えた結果を返します。
s.gsub(t, u)	s に含まれるすべての t を u に置き換えた結果を返します。

表 1.2: 文字列の主要なメソッド

という式を評価すると、レシーバーを文字列のオブジェクトに変換した結果、つまり 7803 という文字列のオブジェクトが得られます。

オブジェクトをあらわす文字列を出力したいときは、display というメソッドを使います。たとえば、

```
8703.display
```

という式を irb に入力したとすると、

```
8703=> nil
```

という文字列が出力されます。ただし、display によって出力された文字列は、この文字列の中の 8703 という部分だけです。その右側の部分は、入力された式の値を示すために irb によって出力されたものです。ちなみに、display の戻り値は、常に nil です。

どんなメソッドも、かならず何らかのオブジェクトを戻り値として返します。しかし、display のように、戻り値を返すことを目的としないメソッドもあります。そのようなメソッドは、普通、意味のない戻り値として nil というオブジェクトを返すように作られています。

次に、display というメッセージを文字列のオブジェクトに送ってみましょう。たとえば、

```
"namako".display
```

という式を評価すると、namako という文字列が出力されます。

改行という文字を含んでいる文字列のオブジェクトを生成したいときは、改行を意味する文字列をリテラルの中に書きます。改行を意味する文字列は、まずバックスラッシュ (\ ) という文字[5]を書いて、その右側に n という文字を書いたものです。たとえば、

```
"umiushi\nhitode"
```

というリテラルを評価すると、改行を含んでいる文字列のオブジェクトが値として得られます。そのような文字列に display というメッセージを送ると、

```
irb(main):001:0> "umiushi\nhitode".display
umiushi
hitode=> nil
```

というように、改行が出力されます。

1.6.2　文字列のメソッド

表 1.2 は、文字列のオブジェクトが持っているメソッドのうちの主要なものを示しています。

文字列の長さ、つまりそれを構成している文字の個数を求めたいときは、文字列のオブジェクトが持っている size というメソッドを使います。たとえば、

```
"umiushi".size
```

[5]バックスラッシュは、日本語の環境では円マーク (¥) で表示されることがあります。

メッセージ式	説明
s.to_i	*s* を整数に変換した結果を返します。
s.to_f	*s* を浮動小数点数に変換した結果を返します。
s.hex	*s* を 16 進数とみなして整数に変換した結果を返します。
s.oct	*s* を 8 進数とみなして整数に変換した結果を返します。

表 1.3: 文字列を数値に変換するメソッド

という式を評価すると、その値として 7 という整数のオブジェクトが得られます。

文字列の末尾にある改行を削除したいときは、chomp というメソッドを使います。たとえば、

```
"kurage\n".chomp
```

という式を評価すると、レシーバーの末尾にある改行を削除することによってできる kurage という文字列のオブジェクトが値として得られます。

文字列から特定の文字を削除したいときは、delete というメソッドを使います。たとえば、

```
"amefurashi".delete("aiueo")
```

という式を評価すると、レシーバーから母音の文字を削除することによってできる mfrsh という文字列のオブジェクトが値として得られます。

1.6.3 文字列を数値に変換するメソッド

表 1.3 に示したメソッドを使うことによって、数値を表現している文字列のオブジェクトを数値のオブジェクトに変換することができます。

10 進数で整数をあらわしている文字列のオブジェクトを整数のオブジェクトに変換したいときは、文字列のオブジェクトが持っている to_i というメソッドを使います。たとえば、

```
"7803".to_i
```

という式を評価すると、7803 という整数のオブジェクトが値として得られます。

文字列のオブジェクトを整数のオブジェクトに変換するメソッドとしては、to_i だけではなくて、hex と oct というメソッドもあります。to_i がレシーバーを 10 進数とみなすのに対して、hex はレシーバーを 16 進数とみなします。たとえば、

```
"ff".hex
```

という式を評価すると、255 という整数のオブジェクトが値として得られます。

oct は、レシーバーを 8 進数とみなして、それを整数のオブジェクトに変換します。ただし、レシーバーが 0b で始まっている場合はそれを 2 進数とみなし、レシーバーが 0x で始まっている場合はそれを 16 進数とみなします。たとえば、

```
"0b10000001".oct
```

という式を評価した場合、レシーバーが 0b で始まっていますので、oct はそれを 2 進数とみなして整数のオブジェクトに変換します。ですから、戻り値は 129 になります。

1.7 Ruby のプログラム

1.7.1 プログラムの実行

Ruby では、「プログラム」(program) という言葉は、「式」(expression) という言葉とほとんど同じ意味だと考えることができます[6]。

Ruby のプログラムは、irb に入力することによって実行することができるわけですが、ファイルの中にプログラムを格納しておいて、それを処理系（irb または ruby）に実行させるということも可能です。

それでは実際に、ファイルの中に格納されているプログラムを処理系に実行させてみましょう。まず、エディターを使って、hello.rb という名前のファイルに次のプログラムを保存してください。

[6]Ruby で書かれたプログラムは、「スクリプト」(script) と呼ばれることもあります。

プログラムの例　hello.rb

```
"みなさん、こんにちは。\n".display
```

なお、Ruby のプログラムを格納するファイルの名前には、このように .rb という拡張子を付けることになっています。

irb を使ってプログラムを実行したいときは、load という関数的メソッドを使います。load は、引数として 1 個の文字列を受け取って、その文字列を、プログラムが格納されているファイルのパス名とみなして、そのプログラムを読み込んで実行します。

それでは、先ほど入力したプログラムを、irb を使って実行してみましょう。

実行例

```
irb(main):001:0> load("hello.rb")
みなさん、こんにちは。
=> true
```

ちなみに、load は、戻り値として常に true を返します。

ruby を使ってプログラムを実行したいときは、シェルに対して、

ruby [パス名]

というコマンドを入力します。そうすると、ruby が起動して、パス名で指定されたファイルの中に格納されているプログラムを実行します。

それでは、先ほど入力したプログラムを、ruby を使って実行してみましょう。

実行例

```
$ ruby hello.rb
みなさん、こんにちは。
```

1.7.2　注釈

人間によって書かれたプログラムというのは、処理系というプログラムによって処理される文書です。プログラムによって処理される文書を書くときには、プログラムがそれを処理できるように書かないといけないのはもちろんですが、それだけではなくて、人間にとって理解しやすいように書くということも重要です。

人間にとって理解しやすい文書を書く上で大きな役割を果たすもののひとつに、「注釈」と呼ばれるものがあります。「注釈」(comment) というのは、文書の中に含まれる、人間がその文書を理解するための手助けとなる記述のことです。

プログラムによって処理される文書の中に注釈を書くためには、この部分は注釈だから処理しなくてもいい、ということがプログラムにわかるような書き方をする必要があります。ですから、プログラムによって処理される文書を書くための言語の中には、たいてい、注釈を書くための規則が含まれています。

Ruby では、「この部分は注釈です」ということを処理系に知らせるために、シャープ (#) という文字を使います。プログラムの中にシャープを書くと、Ruby の処理系は、その直後から改行までの部分を注釈とみなして無視します。

それでは、次のプログラムを入力して、ruby に実行させてみてください。

プログラムの例　comment.rb

```
"Congratulations!\n".display # 日本語では「おめでとう」
```

このプログラムの中にある「日本語では「おめでとう」」という部分は、シャープと改行のあいだに書かれていますので、ruby はその部分を注釈とみなして無視します。さて、それでは、このプログラムからシャープを取り除いて実行すると、いったいどうなるでしょうか。試してみてください。

第 2 章　式

2.1 基本的な式

2.1.1 式についての復習

この章では、式というものについて説明していきたいと思います。そこで、この章を始めるに当たって、まず、第 1 章で式について説明したことを復習しておくことにしましょう。

何らかの動作を記述した文字の列のことを「式」(expression) と呼びます。式が記述している動作を実行することを、式を「評価する」(evaluate) と言います。式を評価すると、その結果として 1 個のオブジェクトが得られます。式を評価した結果として得られるオブジェクトのことを、その式の「値」(value) と呼びます。

特定のオブジェクトを生成するという動作を記述した式のことを「リテラル」(literal) と呼びます。たとえば、7038 というのは、7038 という整数のオブジェクトを生成するリテラルで、"namako" というのは、namako という文字列のオブジェクトを生成するリテラルです。

リテラルを評価すると、それによって生成されたオブジェクトが値として得られます。たとえば、7038 というリテラルを評価すると、その値として、7038 という整数のオブジェクトが得られます。

オブジェクトにメッセージを送る、という動作をあらわしている式のことを「メッセージ式」(message expression) と呼びます。メッセージ式は、基本的には、

式 . メソッド名 (式 , ···)

と書きます。メッセージ式を評価すると、ドット (.) の左側の式を評価することによって得られたオブジェクトに対して、ドットの右側がメッセージとして送られます。すると、メッセージを受け取ったオブジェクト（つまりレシーバー）は、メッセージの中のメソッド名で指定されたメソッドを呼び出して、その右側の式の値を引数としてそのメソッドに渡します。そして、呼び出されたメソッドが戻り値として返したオブジェクトが、メッセージ式全体の値になります。たとえば、

```
"shimauma".gsub("ma", "be")
```

というメッセージ式を評価すると、"shimauma" という文字列のオブジェクトの中の gsub というメソッドが呼び出されて、"ma" と "be" が引数としてそのメソッドに渡されます。gsub は、レシーバーの中の "ma" を "be" に置き換えた結果を戻り値として返しますので、"shibeube" という文字列が、そのメッセージ式の値として得られることになります。

2.1.2 式列

式を並べることによってできる列のことを「式列」(expression sequence) と呼びます。ただし、式列を構成するそれぞれの式は、セミコロン (;) または改行で区切られていないといけません。つまり、式列というのは、

式 ; 式 ; ···

という形で式を並べたもの、または、

式
式
⋮

という形で式を並べたもの、ということになります。

式列は、その全体がひとつの式になります。式列を評価すると、それを構成するそれぞれの式が、先頭から末尾に向かって順番に評価されていきます。たとえば、

```
"mike".display; "neko".display; "\n".display
```

という式列を評価すると、まず mike という文字列が出力されて、次に neko という文字列が出力されて、そののち改行が出力されます。irb で試してみると、次のようになります。

```
irb(main):001:0> "mike".display; "neko".display; "\n".display
mikeneko
=> nil
```

それでは、次のプログラムをファイルに保存してください。

プログラムの例 expsequ.rb

```
"リテラル\n".display
"メッセージ式\n".display
"式列\n".display
```

プログラムの保存ができたら、そのプログラムを実行してみてください。そうすると、式列の中の式が順番に評価されて、次のように出力されるはずです。

実行例

```
irb(main):001:0> load("expsequ.rb")
リテラル
メッセージ式
式列
=> true
```

式列を評価すると、その末尾の式の値が、式列全体の値として得られます。つまり、

$\boxed{式_1}$; $\boxed{式_2}$; ···; $\boxed{式_n}$

という式列を評価すると、その末尾にある、式 $_n$という式の値が、式列全体の値になります。ですから、末尾以外の式は、その値が失われることになります。irb で試してみると、次のようになります。

```
irb(main):001:0> "kitsune"; 604; "tanuki"; 532
=> 532
```

2.2 リテラル

2.2.1 リテラルについての復習など

すでに第 1.3 節で説明したように、特定のオブジェクトを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。たとえば、"suzume" のような、文字列を二重引用符で囲んだものは、リテラルの一種です。

リテラルを評価すると、それによって生成されたオブジェクトが、その値として得られます。たとえば、"suzume" というリテラルを評価すると、それによって生成された suzume という文字列のオブジェクトが、その値として得られます。

第 1.3 節でのリテラルについての説明は、きわめて不十分なものでしたので、この節では、リテラルについて、もう少し詳細な説明を加えたいと思います。

ところで、この文章の中では、これまでのところ、文字列を取り扱うさまざまなメソッドを持っているオブジェクトのことを「文字列のオブジェクト」と呼んできました。しかし、これから先は、文字列のオブジェクトを指していることが文脈から明らかに分かる場合には、「文字列のオブジェクト」のことを単に「文字列」と呼ぶことにします。

同じように、文字列以外のオブジェクトについても、誤解のおそれがなければ、「○○のオブジェクト」のことを単に「○○」と呼ぶことにします。

2.2.2 整数のリテラル

481 とか 3007 というような、数字だけから構成される列は、プラスの整数のオブジェクトを生成するリテラルになります。たとえば、481 というリテラルを評価すると、481 というプラスの整数のオブジェクトが値として得られます。

整数のオブジェクトは、Fixnum というクラスのインスタンスです。ただし、巨大な整数のオブジェクトは、Fixnum ではなくて Bignum というクラスのインスタンスです。

それでは、irb で、整数のオブジェクトを生成したクラスを調べてみましょう。

```
irb(main):001:0> 10.class
=> Fixnum
irb(main):002:0> 100000000000000000000000000000.class
=> Bignum
```

ちなみに、Fixnum と Bignum の境目は、Ruby の仕様ではなくて、環境に依存して決定されます。

数字の列の左端にマイナス (-) という文字を書くと、その全体は、マイナスの整数を生成するリテラルになります。たとえば、 -56 というリテラルは、マイナスの 56 という整数を生成します。

10 進数ではなくて、8 進数や 16 進数や 2 進数で整数を生成することも可能です。10 進数以外の基数で整数を生成したいときは、そのための接頭辞をリテラルの先頭に書きます。接頭辞は、8 進数は 0 、16 進数は 0x 、2 進数は 0b です。

たとえば、 0377 、 0xff 、 0b11111111 は、いずれも、255 という整数を生成します。

2.2.3 浮動小数点数のリテラル

ひとつの数値を、数字の列と小数点の位置という二つの要素で表現しているデータは、「浮動小数点数」(floating point number) と呼ばれます。浮動小数点数のオブジェクトは、 Float というクラスのインスタンスです。

0.003 とか 41.56 とか 723.0 というような、数字の列の途中に、ドット (.) という文字を 1 個だけ書いたものは、プラスの浮動小数点数のオブジェクトを生成するリテラルになります。この場合、ドットは小数点の位置を示します。

整数のリテラルの場合と同じように、浮動小数点数のリテラルの場合も、左端にマイナス (-) という文字を書くと、その全体は、マイナスの浮動小数点数を生成するリテラルになります。たとえば、 -8.317 というリテラルは、マイナスの 8.317 という浮動小数点数を生成します。

浮動小数点数のリテラル (a) の右側に e という文字を書いて、そのさらに右側に整数のリテラル (b) を書いた、

$$a\,\mathtt{e}\,b$$

という形のリテラルを書くことも可能です。この形のリテラルを評価すると、

$$a \times 10^b$$

という浮動小数点数が生成されます。たとえば、

```
-3.71e-24
```

というリテラルは、

$$-3.71 \times 10^{-24}$$

という浮動小数点数を生成します。

2.2.4 文字列のリテラル

第 1.3 節で説明したように、文字列を二重引用符 (") で囲んだものは、その文字列のオブジェクトを生成するリテラルになります。たとえば、 "namako" というのは、 namako という文字列のオブジェクトを生成するリテラルです。

2.2.5 バックスラッシュ記法

改行のような、出力装置を制御するための特殊な文字は、文字列のリテラルの中では、「バックスラッシュ記法」(backslash notation) と呼ばれる表記法によって記述されます。

バックスラッシュ記法というのは、バックスラッシュ (\ ) という文字で始まる文字列によって 1 個の文字を記述する表記法のことです。たとえば、バックスラッシュ記法を使うことによって、タブは \t 、改行は \n 、改ページは \f と記述することができます。ですから、

```
isoginchaku
fujitsubo
```

というような、改行を含んでいる文字列は、

```
"isoginchaku\nfujitsubo"
```

というリテラルを書くことによって生成することができます。

バックスラッシュ記法は、出力装置を制御する文字を記述するときだけではなくて、文字列のリテラルの中で特殊な意味を持つ文字を記述するときにも使われます。たとえば、二重引用符 (") という文字は、文字列のリテラルを囲むために使われる文字ですので、文字列のリテラルの中で二重引用符を記述したいときは、 \" と書く必要があります。そして、バックスラッシュという文字も特殊な意味を持っていますので、文字列のリテラルの中でバックスラッシュを記述したいときは、 \\ と書く必要があります。

2.2.6 式展開

文字列のリテラルの中に式を書いておいて、文字列のリテラルが評価された時点で、その式を評価して、その値を文字列の中に埋め込む、ということが可能です。

しかし、式の値を文字列に埋め込みたいからと言って、ただ単に文字列のリテラルの中に式を書いただけでは、その意図が処理系には伝わりません。式の値を文字列に埋め込みたいという意図を処理系に伝えるためには、単なる式ではなくて、「式展開」(expression unfolding) と呼ばれる文字列を書く必要があります。

式展開は、

```
#{式}
```

という構文を持つ文字列です。たとえば、

```
"\"subject\" consists of #{"subject".size} characters."
```

というリテラルを評価すると、その中に書かれた式がその時点で評価されて、

```
"subject" consists of 7 characters.
```

という文字列が生成されます。

2.2.7 文字のリテラル

文字のオブジェクトは、Fixnum クラスのインスタンスです。つまり、文字は整数によってあらわされるわけです。たとえば、大文字の A は 65 という整数であらわされ、スラッシュ(/) は 47 という整数であらわされます。文字をあらわしている整数は、その文字の「文字コード」(character code) と呼ばれます[1]。

ですから、特定の文字のオブジェクトを生成したいときは整数のリテラルを書けばいい、ということになるわけですが、その方法は少し不便です。実は、整数を生成するリテラルには、まだ説明していないもうひとつの書き方があって、特定の文字のオブジェクトを生成したいときは、その書き方を使うと便利です。

整数を生成するリテラルのもうひとつの書き方というのは、まずクエスチョンマーク (?) を書いて、その右側に 1 個の文字を書く、というものです。そのようなリテラルを評価すると、クエスチョンマークの右側に書かれた文字に対応する文字コードのオブジェクトが生成されます。たとえば、?M というリテラルは、大文字の M の文字コード (77) を生成します。

クエスチョンマークの右側に、バックスラッシュ記法で文字を記述することも可能です。たとえば、?\n というリテラルは、改行という文字の文字コード (10) を生成します。

2.3 演算子

2.3.1 演算

メッセージ式とは異なる形の式を書くことによって呼び出すことのできる動作のことを「演算」(operation) と呼びます。演算というのは、基本的にはメソッドの一種だと考えていいのですが、メソッドではない演算というのもいくつか存在します。

演算に与えられた名前のことを「演算子」(operator) と呼びます。そして、演算を呼び出すために書かれる、メッセージ式とは異なる形の式は、「演算子式」(operator expression) と呼ばれます。演算子式にはいくつかの形があって、どの形の演算子式を書けばいいのかというのは、呼び出したい演算の種類によって決まります。

二つのオブジェクトを処理の対象とする演算は「二項演算」(binary operation) と呼ばれ、それに与えられた名前は「二項演算子」(binary operator) と呼ばれます。二項演算を呼び出したい場合は、

式 二項演算子 式

という形の演算子式を書きます。

ひとつのオブジェクトを処理の対象とする演算は「単項演算」(unary operation) と呼ばれ、それに与えられた名前は「単項演算子」(unary operator) と呼ばれます。単項演算を呼び出したい場合は、

[1]「文字コード」という言葉は、文字と整数とを対応させる規則、という意味で使われることもあります。

演算子式	説明
a + b	a と b とを加算した結果を返します。
a - b	a から b を減算した結果を返します。
a * b	a と b とを乗算した結果を返します。
a ** b	a の b 乗を返します。
a / b	a を b で除算したときの商を返します。
a % b	a を b で除算したときのあまりを返します。
+ a	a をそのまま返します。
- a	a の符号（プラスかマイナスか）を反転させた結果を返します。

表 2.1: 算術演算

単項演算子 式

という形の演算子式を書きます。

2.3.2 算術演算

数値（整数または浮動小数点数）のオブジェクトは、加算、減算、乗算、除算などを実行する演算のメソッドを持っています。それらの演算は、総称して「算術演算」(arithmetic operation) と呼ばれ、それらの演算の名前は「算術演算子」(arithmetic operator) と呼ばれます。算術演算には、表 2.1 のようなものがあります。

それでは、実際に算術演算を呼び出してみましょう。まず、

```
7.+(8)
```

というメッセージ式を irb に入力してみてください。すると、15 という値が得られるはずです。

+ というのは演算子ですから、メッセージ式ではなくて演算子式を書くことによって呼び出すことも可能です。そこで次に、

```
7+8
```

という演算子式を入力してみてください。すると、先ほどと同じ結果になるはずです。

2.3.3 範囲演算

始端と終端を意味する二つのオブジェクトによって指定されるオブジェクトの列は、「範囲」(range) と呼ばれます。範囲のオブジェクトは、Range というクラスのインスタンスです。

範囲のオブジェクトは、「範囲演算」(range operation) と呼ばれる二項演算を呼び出すことによって生成することができます。

範囲演算には、.. と ... の 2 種類があります。

.. は、始端と終端の両方を含む範囲のオブジェクトを生成する演算で、... は、始端は含むけれども終端は含まない範囲のオブジェクトを生成する演算です。たとえば、

```
30..80
```

という式を評価すると、30 から 80 までの範囲が生成され、

```
30...80
```

という式を評価すると、30 から 79 までの範囲が生成されます。

整数だけではなくて、文字列の範囲を生成することも可能です。たとえば、

```
"ax".."bc"
```

という式を評価すると、

```
"ax" "ay" "az" "ba" "bb" "bc"
```

という範囲が生成されます。

演算子式	説明
s + t	s の右側に t を連結した結果を返します。
s * n	s を n 回繰り返して連結した結果を返します。
s[i]	s の i 番目の文字の文字コードを返します。
s[i..j]	s の i 番目から j 番目までの部分文字列を返します。
s[i, n]	s の i 番目から始まる長さが n の部分文字列を返します。

表 2.2: 文字列演算

2.3.4 文字列演算

文字列のオブジェクトも、「文字列演算」(string operation) と呼ばれるいくつかの演算を持っています。表 2.2 は、文字列演算のうちの主要なものを示しています。

文字列の + は、文字列と文字列とを連結します。たとえば、

```
"kitsune" + "udon"
```

という式を評価すると、kitsuneudon という文字列が値として得られます。

文字列の * は、ひとつの文字列を指定された回数だけ繰り返して連結します。たとえば、

```
"----+" * 12
```

という式を評価すると、

```
----+----+----+----+----+----+----+----+----+----+----+----+
```

という文字列が値として得られます。

角括弧 ([]) という演算を呼び出すための式は、これまでに出てきた演算子式とは少し形が違いますが、これもまた演算子式の一種です。角括弧の左側の式の値がレシーバーになって、角括弧で囲まれた場所に書かれた式の値が引数になります。

文字列を構成するそれぞれの文字には、先頭が 0 番、次の文字が 1 番、というように番号が与えられています。[] に引数として整数を渡すと、[] は、その整数を番号とする文字をレシーバーの中から取り出して、その文字を戻り値として返します。

それでは、実際に [] を呼び出してみましょう。たとえば、

```
"namako"[4]
```

という式を評価すると、107 という整数が値として得られます。この 107 という整数は、k という文字の文字コードです。

[] に対して、引数として範囲を渡すと、[] は、受け取った範囲によって指定される部分文字列を戻り値として返します。たとえば、

```
"isoginchaku"[3..8]
```

という式を評価すると、3 番目から 8 番目までの部分文字列、つまり gincha という文字列が値として得られます。

s[i, n] という形の式で [] を呼び出すと、[] は、i 番目から始まる長さが n の部分文字列を戻り値として返します。たとえば、

```
"tatsunootoshigo"[7, 6]
```

という式を評価すると、7 番目から始まる長さが 6 の部分文字列、つまり otoshi という文字列が値として得られます。

2.3.5 優先順位と結合規則

ところで、$\circ$ と $\bullet$ のそれぞれが二項演算子で、a と b と c のそれぞれが式だとするとき、

$$a \circ b \bullet c$$

という式は、

$$\boxed{a \circ b} \bullet c$$

と解釈されるのでしょうか、それとも、

`[]` `[]=`
`**`
単項の `+` 単項の `-` `!` `~`
`*` `/` `%`
二項の `+` 二項の `-`
`<<` `>>`
`&`
`|` `^`
`>` `<` `>=` `<=`
`<=>` `==` `===` `!=` `=~` `!~`
`&&`
`||`
`..` `...`
`?` `:`
`=` 自己代入演算子
`defined?`
`not`
`and` `or`
`if` `unless` `while` `until`
`begin/end`

表 2.3: 演算子の優先順位

$$a \circ \boxed{b \bullet c}$$

と解釈されるのでしょうか。

そのような、演算子が何個も含まれている式は、それぞれの演算子が持っている、「優先順位」(precedence) という属性か、または「結合規則」(associativity) という属性にしたがって解釈されます。

優先順位というのは、演算子が左右の式と結合する強さの順位だと考えることができます。優先順位の高い演算子は、低い演算子に比べて、より強く左右の式と結合します。

表 2.3 は、演算子の優先順位を示したものです（まだ説明していない演算子がたくさん含まれていますが、その点は気にしないでください）。この表は、上のほうに書かれている演算子ほど高い優先順位を持っていて、横に並べて書かれている演算子は同一の優先順位を持っているということを意味しています。

たとえば、乗算と除算の演算子は、加算と減算の演算子よりも高い優先順位を持っていますので、

$$a + b * c$$

という式は、

$$a + \boxed{b * c}$$

と解釈されます。

さて、それでは、同一の優先順位を持つ演算子がいくつも含まれている、

$$a - b + c$$

というような式は、どのように解釈されるのでしょうか。

そのような式は、演算子の優先順位では解釈できませんので、演算子が持っている結合規則という属性によって解釈されることになります。結合規則は、同一の優先順位を持つ演算子が共有している性質です。それらの演算子がひとつの式の中に何個も含まれているとき、それぞれの演算子が左右の式と結合する強さは、それらの演算子が共有している結合規則によって決定されま

す。左にあるものほど強くなるという結合規則は「左結合」(left-associativity) と呼ばれ、右にあるものほど強くなるという結合規則は「右結合」(right-associativity) と呼ばれます。

これまでに登場した二項演算子の結合規則は、すべて左結合です。したがって、

```
a - b + c
```

という式は、

```
[a - b] + c
```

と解釈されることになります。

2.3.6 丸括弧式

丸括弧で式を囲んだもの、つまり、

```
( [   式   ] )
```

という形のものは、「丸括弧式」(parenthesis expression) と呼ばれる式になります。丸括弧式を評価すると、丸括弧の中の式が評価されて、その値が丸括弧式全体の値になります。

丸括弧式は、優先順位や結合規則とは無関係に、式を書く人間の意図のとおりに式を解釈してほしい、という場合に使われます。たとえば、

```
[a + b] * c
```

と解釈されるような式を書きたい、というときは、

```
(a + b) * c
```

というように丸括弧式を使えばいいわけです。

メッセージ式を書くときに使われるドット (.) というのは、厳密には演算子ではありませんが、式の解釈という観点から見た場合、優先順位のきわめて高い演算子だと考えることができます。ですから、○が二項演算子だとするとき、

```
a ○ b . message
```

という式は、

```
a ○ [b . message]
```

と解釈されることになります。このような、ドットを含んでいる式に関しても、丸括弧式を使うことによって、どう解釈してほしいのかということを指定することができます。たとえば、

```
[a ○ b] . message
```

と解釈されるような式を書きたい、というときは、

```
(a ○ b) . message
```

というように丸括弧式を使えばいいわけです。

2.4 変数

2.4.1 変数の基礎

プログラミング言語の多くは、「変数」(variable) と呼ばれるものを取り扱います。この「変数」という言葉の意味は、プログラミング言語ごとに微妙に異なります。Ruby でも「変数」と呼ばれるものを取り扱うのですが、Ruby の「変数」は、オブジェクトに付ける名札のようなもの、という意味です。

プログラムの多くは、リテラルで記述されるような特定のオブジェクトだけではなくて、特定できないオブジェクトというものも扱います。そのような不特定のオブジェクトを扱うためには、それに名札を付けることが必要になります。オブジェクトに対して変数という名札を付けておくと、それの実体が何であるかにかかわらず、それをプログラムの中で取り扱うことができるようになります。

オブジェクトに変数という名札が付けられているとき、その変数はそのオブジェクトを「指し示す」(point) と言われます。

変数には、ローカル変数 (local variable)、グローバル変数 (global variable)、インスタンス変数 (instance variable)、クラス変数 (class variable) という 4 種類のものがあります。ローカル変数というのがごく普通の変数のことで、グローバル変数、インスタンス変数、クラス変数というのは、やや特殊な変数です。なお、この節では、ローカル変数のことをただ単に「変数」と呼ぶことにします。

2.4.2 変数名

変数という名札には、ひとつの名前が書かれています。変数に書かれている名前のことを、「変数名」(variable name) と呼びます。変数名は、英字、数字、アンダースコア (_) を並べることによって作ります。ただし、変数名の先頭の文字は英字の小文字でないといけません。

たとえば、`a`、`namako`、`uni83`、`ika_tako`、`ikaTako`、などは、変数名として使うことのできる名前です。

変数名は、式として評価することができます。変数名を評価することによって得られる値は、その変数が指し示しているオブジェクトです。たとえば、`namako` という名前の変数が、684 というオブジェクトに名札として与えられているとするとき、`namako` という変数名を評価すると、684 というオブジェクトが値として得られます。

2.4.3 代入

オブジェクトに変数という名札を付けることを、変数にオブジェクトを「代入する」(assign) と言います。

変数にオブジェクトを代入したいときは、「代入演算子」(assignment operator) と呼ばれるいくつかの演算子のうちのいずれかを使います。代入演算子は、変数にオブジェクトを代入するという動作をあらわしていて、その動作は「代入演算」(assignment operation) と呼ばれます（代入演算はメソッドではありません)。

代入演算子のうちで、もっとも単純な演算をあらわしているのは、`=` という演算子です。この演算子は、左辺で指定された変数に右辺の値を代入する、という演算をあらわしています。たとえば、

```
namako = 4004
```

という式を評価することによって、`namako` という変数に 4004 という整数のオブジェクトを代入することができます（`namako` という変数が存在していない場合は、この時点で新しく作られます)。

変数が指し示すオブジェクトは、何度でも変更することが可能です。変数が指し示すオブジェクトを変更したいときは、ただ単に、別のオブジェクトを代入すればいいだけです。たとえば、今、4004 というオブジェクトを指し示している `namako` という変数があるとしましょう。このとき、

```
namako = 3773
```

という式で `namako` に 3773 を代入したとすると、`namako` が指し示すオブジェクトは、4004 から 3773 へ変更されます。

それでは、irb を使って、実際に変数にオブジェクトを代入してみましょう。

```
irb(main):001:0> namako = 4004
=> 4004
irb(main):002:0> namako
=> 4004
irb(main):003:0> namako = 3773
=> 3773
irb(main):004:0> namako
=> 3773
```

2.4.4 複数の変数への同一オブジェクトの代入

Ruby では、二項演算子の大多数は結合規則が左結合ですが、代入演算子の結合規則は右結合です。つまり、

$$a = b = c$$

という式は、

a = *b* = *c*

と解釈されることになります。

代入演算子を使って変数にオブジェクトを代入する式を評価すると、その値として、代入ののちに変数が指し示すことになったオブジェクトが得られます。

ところで、変数にオブジェクトを代入する式は、代入という動作が第一の目的ですので、式の値を利用するということはめったにありません。しかし、利用すれば便利だという場面が、まったくないわけではありません。

そのような場面のひとつとして、複数の変数に同一のオブジェクトを代入したい、というときがあります。変数にオブジェクトを代入する式の値を利用すると、複数の変数に同一のオブジェクトを代入するという動作をひとつの式で書くことが可能になります。たとえば、

```
a = b = c = d = 3579
```

という式を書くことによって、4 個の変数のそれぞれに 3579 という同一の整数のオブジェクトを代入することができます。

2.4.5 自己代入演算子

= 以外のすべての代入演算子は、「自己代入演算子」(self assignment operator) と呼ばれるものです。自己代入演算子というのは、変数が指し示しているオブジェクトに対して演算を実行することによって得られたオブジェクトを、元の変数に代入する、という動作をあらわしている演算子のことです。

自己代入演算子は、すべて、*op*= という形になっています。この中の *op* のところには、+、-、*、/ などの二項演算子が入ります。二項演算子とイコールとのあいだに空白を入れることはできません。

自己代入演算子を使って、

変数名 *op*= 式

という式を書いたとすると、それは、

変数名 = 変数名 *op* 式

という意味だと解釈されます。たとえば、

```
a += 3
```

という式は、

```
a = a + 3
```

という式と同じ意味になります。つまり、a という変数が指し示しているオブジェクトを、それを 3 だけ大きくしたオブジェクトに変更するわけです。

2.5 メソッドの定義

2.5.1 メソッドの定義の基礎

新しいメソッドを作ることを、メソッドを「定義する」(define) と言います。

メソッドを定義するためには、その新しいメソッドに何らかの名前を付ける必要があります。メソッドの名前は、変数の名前と同じように、英字、数字、アンダースコア (_) を並べることによって作ります。そして、先頭の文字は、英字の小文字でないといけません。たとえば、namako、uni587、ika_tako、ikaTako などは、メソッドに与えることのできる名前です。

メソッドを定義したいときは、「メソッド定義」(method definition) と呼ばれる式を書きます。

引数を受け取らないメソッドは、

```
def メソッド名
  式
end
```

という形のメソッド定義によって定義することができます。この形の式を評価すると、新しいメソッドが作られて、そのメソッドに対して、「メソッド名」のところに書かれた名前が与えられ

ます。そのとき、メソッド定義の中に書かれている式は、評価されません。
メソッドは、そのメソッドを定義したメソッド定義の中に書かれている式を評価する、という動作をします。つまり、メソッドを呼び出すと、そのたびに、そのメソッドを定義したメソッド定義の中に書かれている式が評価されることになります。

2.5.2 irb へのメソッド定義の入力

第 1.4 節で説明したように、irb に入力した式は、「トップレベル」(toplevel) と呼ばれる文脈で評価されます。レシーバーを省略したメッセージ式をトップレベルで評価すると、そのメッセージは、「トップレベルオブジェクト」(toplevel object) と呼ばれるオブジェクトに送られます。
ところで、メソッドというのは、かならず何らかのオブジェクトの中に存在するものです。それでは、メソッド定義をトップレベルで評価した場合、新しいメソッドはどこに作られるのでしょうか。その答は、「トップレベルオブジェクトの中」です。
それでは、irb にメソッド定義を入力して、新しいメソッドをトップレベルオブジェクトの中に作ってみましょう。まず、

```
def kujira
  "kujira\n".display
end
```

という式を irb に入力してください。irb は、式の入力の途中で改行が入力された場合は、

```
irb(main):001:0> def kujira
irb(main):002:1> 
```

というように、式の続きの入力を促すプロンプトを出力します。そして、式を最後まで入力すると、

```
irb(main):001:0> def kujira
irb(main):002:1>   "kujira\n".display
irb(main):003:1> end
=> nil
```

というように、その式の値を出力します。ちなみに、メソッド定義の値は、常に `nil` です。
さて、これで、`"kujira\n"` という文字列を出力する、`kujira` というメソッドがトップレベルオブジェクトの中に作られたはずですので、次に、それを呼び出してみましょう。
トップレベルでは、レシーバーが書かれていないメッセージ式は、トップレベルオブジェクトにメッセージを送る式だと解釈されます。ということは、トップレベルオブジェクトの中にあるメソッドをトップレベルで呼び出したいときは、レシーバーを書かずに、ただ単にメッセージだけを書けばいい、ということになります。
試してみましょう。irb に対して、

```
kujira
```

というメッセージ式を入力してください。そうすると、`kujira` が呼び出されて、`"kujira\n"` という文字列が出力されるはずです。

2.5.3 ファイルに保存されたメソッド定義

今度は、メソッド定義をファイルに保存して、それを irb に実行させてみましょう。まず、次のプログラムを入力して、ファイルに保存してください。

プログラムの例 `fortune.rb`

```
def fortune
  "あなたの今日の運勢は大吉です。\n".display
end
```

次に、irb に対して、

```
load("fortune.rb")
```

という式を入力してください。すると、プログラムが実行されるわけですが、その中で定義されている `fortune` というメソッドは、いったいどこに作られるのでしょうか。
プログラムの地肌に相当する場所（つまり、そこを囲んでいる枠組みが何もない場所）に書かれた式が評価される文脈は、やはりトップレベルです。したがって、`fortune` は、トップレベル

オブジェクトの中に作られることになります。ですから、そのメソッドは、レシーバーを省略したメッセージ式を irb に入力することによって呼び出すことができます。それでは実際に、

```
fortune
```

というメッセージ式を irb に入力してみましょう。そうすると、「あなたの今日の運勢は大吉です。」という託宣が出力されるはずです。

2.5.4 インデント

メソッド定義という式は、枠組みの中に式を書く、という構造になっています。このような構造の式を書くときは、普通、枠組みの中の式を少しだけ右にずらして書きます。その理由は、そのほうがプログラムを読む人間にとって式の構造が理解しやすくなるからです。

行の先頭に空白を入れることによって式を右にずらすことを、式を「インデントする」(indent) と言います。Ruby では、2 個の空白を使って式をインデントするというのが慣習になっています。

2.5.5 クラス定義

トップレベルオブジェクトの中にメソッドを作るのではなくて、整数や文字列などのオブジェクトの中にメソッドを作りたいときは、「クラス定義」(class definition) と呼ばれる式の中にメソッド定義を書く必要があります。

クラス定義というのは、

```
class クラス名
  式
end
```

という形の式です。クラス定義の中にはどんな式を書いてもかまわないのですが、普通はそこにメソッド定義を書きます。

クラス定義を評価すると、その過程で、その中に書かれている式も評価されます。クラス定義の中に書かれている式は、クラス定義の文脈で評価されます。

クラス定義の文脈でメソッド定義を評価すると、そのメソッド定義によって定義されるメソッドを作るための鋳型が、クラス定義の冒頭で指定されたクラスに追加されます。

クラスという鋳型は、さまざまなメソッドの鋳型から構成されています。クラスから生成されたインスタンスは、そのクラスが持っているメソッドの鋳型から作られたメソッドを持つことになります。ですから、新しいメソッドの鋳型をクラスに追加するということは、そのクラスのインスタンスに新しいメソッドを追加するということを意味します。

それでは、実際に試してみましょう。まず、次のプログラムを入力して、ファイルに保存してください。

プログラムの例 `strwhat.rb`

```
class String
  def what
    "I am a string.\n".display
  end
end
```

次に、irb に対して、

```
load("strwhat.rb")
```

という式を入力してください。すると、`what` というメソッドを作るための鋳型が `String` クラスの中に作られます。ということは、`String` クラスのインスタンスは `what` というメソッドを持つということですから、

```
"namako".what
```

というように、文字列に `what` というメッセージを送ると、

```
I am a string.
```

という文字列が出力されることになります。

ちなみに、存在しないクラスの名前を指定したクラス定義を評価した場合は、その名前を持つ新しいクラスが作られて、その新しいクラスにメソッドの鋳型が追加されます。クラスを作るこ

とを、クラスを「定義する」(define) と言います。クラスの定義については、第 3.1 節で説明することにしたいと思います。

2.5.6 継承

第 1.5 節で、スーパークラスとサブクラスというものについて説明しました。それは、次のようなクラス間の関係のことでした。

A というクラスがあって、それを特殊化することによって B というクラスを作ったとするとき、A のことを、B の「スーパークラス」(superclass) と言い、B のことを、A の「サブクラス」(subclass) と言います。たとえば、`Fixnum` というクラスは `Integer` というクラスを特殊化することによって作られたものですので、`Integer` は `Fixnum` のスーパークラスで、`Fixnum` は `Integer` のサブクラスだということになります。

スーパークラスとサブクラスについては、第 1.5 節で説明したことに加えて、さらに、「継承」(inheritance) と呼ばれるものについて理解しておく必要があります。

クラスを定義するというのは、常に、何らかのクラスをスーパークラスとするサブクラスとして新しいクラスを作るということです。そのとき、新しく作られたクラスは、そのスーパークラスから、それが持っている性質をそのまま受け継ぎます。「継承」というのは、サブクラスがスーパークラスの性質を受け継ぐことです。

スーパークラスからサブクラスへ継承される性質というのは、どんなメソッドの鋳型を持っているかということだと考えることができます。つまり、スーパークラスが持っているメソッドの鋳型は、すべてサブクラスへ継承されることになるわけです。

クラスの階層を上へ上へとたどっていくと、かならず `Object` というクラスにたどりついて、そこで行き止まりになります。つまり、`Object` というクラスは、クラスの木の根の位置にあるわけです。ですから、`Object` クラスにメソッドの鋳型を追加すると、その鋳型はすべてのクラスへ継承されます。つまり、すべてのオブジェクトがそのメソッドを持つことになるということです。

それでは、次のプログラムをファイルに保存して、irb に実行させてください。

プログラムの例 `objwhat.rb`

```
class Object
  def what
    "I am an object.\n".display
  end
end
```

そうすると、`Object` クラスに `what` というメソッドの鋳型が追加され、その鋳型はすべてのクラスへ継承されます。つまり、すべてのオブジェクトが `what` というメソッドを持つことになるわけです。ですから、

```
7614.what
```

というように整数のオブジェクトに `what` というメッセージを送ると、

```
I am an object.
```

という文字列が出力されるはずです。

2.5.7 レシーバーの文脈

メソッド定義の中に書かれている式は、レシーバーの文脈で評価されます。レシーバーの文脈というのは、「レシーバーを省略したメッセージ式は、レシーバー自身にメッセージを送るものだと解釈される」という文脈のことです。ここで「レシーバー自身」と呼ばれているのは、実行中のメソッドを含んでいるオブジェクトのことです。

第 1.4 節で説明したように、オブジェクトがメッセージを受け取ると、そのオブジェクトは、受け取ったメッセージにしたがって自分の中にあるメソッドに仕事をさせます。そのとき、そのメソッドは、自分の仕事を遂行するために、自分と同じオブジェクトの中にある別のメソッドを呼び出すことがしばしば必要になります。

メソッド定義の中では、「自分と同じオブジェクトの中にある別のメソッドを呼び出す」という動作は、レシーバーを省略したメッセージ式によって記述することができます。なぜなら、そのメッセージ式はレシーバーの文脈で評価されるからです。

それでは、次のプログラムをファイルに保存して、irb で実行してください。

プログラムの例 twice.rb

```
class String
  def twice
    display
    display
    "\n".display
  end
end
```

twiceというメソッドを定義する式の中には、displayというメッセージが3個書かれていますが、上の二つはレシーバーが指定されていませんので、twiceというメッセージを受け取ったレシーバー自身に送られます。ですから、

```
"Manami".twice
```

という式で、"Manami"という文字列にtwiceというメッセージを送ったとすると、

```
ManamiManami
```

というように、レシーバーの出力が2回繰り返されることになります。

2.5.8 レシーバー自身を求める式

メソッド定義の中では、しばしば、レシーバー自身を求める必要が生じます。レシーバー自身を求めたいときは、selfという式を書きます。selfを評価すると、その値としてレシーバー自身が得られます。

次のプログラムは、2のレシーバー乗を出力する、powerOfTwoというメソッドを定義しています。

プログラムの例 powtwo.rb

```
class Numeric
  def powerOfTwo
    (2 ** self).display
    "\n".display
  end
end
```

実行例

```
irb(main):001:0> load("powtwo.rb")
=> true
irb(main):002:0> 8.powerOfTwo
256
=> nil
irb(main):003:0> 0.5.powerOfTwo
1.414213562
=> nil
```

ちなみに、このプログラムに登場しているNumericというのは、IntegerとFloatのスーパークラスです。Numericにメソッドの鋳型を追加すると、その鋳型は、IntegerとFloatの両方へ継承されます。ですから、整数と浮動小数点数の両方に同じメソッドを追加したいときは、クラスとしてNumericを指定すればいいわけです。

2.5.9 引数

引数を受け取るメソッドを定義したいときは、

```
def メソッド名(仮引数名, ··· )
  式
end
```

という形のメソッド定義を書きます。「仮引数名」というところには、英字の小文字で始まる名前を書きます。

「仮引数名」のところに書かれた名前は、「仮引数」(formal argument) と呼ばれるものに与えられます。仮引数というのは、引数を受け取るための変数のことです。

どの仮引数がどの引数を受け取ることになるかというのは、それらが書かれている順番で決まります。たとえば、

```
def namako(a, b, c)
  式
end
```

と定義されたメソッドを、

```
namako(24, 33, 81)
```

というメッセージで呼び出したとすると、 a が 24 を、 b が 33 を、 c が 81 を受け取ることになります。

次のプログラムの中で定義されている rect というメソッドは、縦の個数と横の個数を引数として受け取って、文字列のレシーバーを長方形の形に並べて出力します。

プログラムの例 rect.rb

```
class String
  def rect(tate, yoko)
    ((self * yoko + "\n") * tate).display
  end
end
```

実行例

```
irb(main):001:0> load("rect.rb")
=> true
irb(main):002:0> "Mikako".rect(4, 6)
MikakoMikakoMikakoMikakoMikakoMikako
MikakoMikakoMikakoMikakoMikakoMikako
MikakoMikakoMikakoMikakoMikakoMikako
MikakoMikakoMikakoMikakoMikakoMikako
=> nil
```

2.5.10 戻り値

メソッドは、自分を定義したメソッド定義の中に書かれている式の値を、戻り値として返します。たとえば、 iruka というメソッドが、

```
def iruka
  2222
end
```

と定義されているとすると、 iruka は、常に 2222 を戻り値として返すことになります。

次のプログラムは、数値のレシーバーを 2 乗した結果を戻り値として返す、 square というメソッドを定義します。

プログラムの例 square.rb

```
class Numeric
  def square
    self * self
  end
end
```

実行例

```
irb(main):001:0> load("square.rb")
=> true
irb(main):002:0> 7.square
=> 49
```

次のプログラムは、1 からレシーバーまでの整数の和を戻り値として返す、 sumOfNumbers というメソッドを定義します。

プログラムの例 sumnum.rb

```
class Integer
```

```
  def sumOfNumbers
    self * (self + 1) / 2
  end
end
```

実行例

```
irb(main):001:0> load("sumnum.rb")
=> true
irb(main):002:0> 10.sumOfNumbers
=> 55
irb(main):003:0> 100.sumOfNumbers
=> 5050
```

2.5.11 メソッドを定義する必要性

プログラムというものは、コンピュータに理解できればそれでいいというものではなくて、人間にとっても理解しやすいように書く必要があります。しかし、プログラムによって記述されている動作が複雑になればなるほど、そのプログラムは、それを読む人間にとって理解することが困難なものになっていきます。ですから、複雑な動作をするプログラムを書く場合は、それを少しでも人間にとって理解しやすいものにするための工夫が必要です。

複雑な動作をわかりやすく記述するための工夫としてもっとも効果があるのは、いきなり全体を記述するのではなくて、全体をいくつかの部分に分割して、それぞれの部分を、その内部の構造を知らなくても使うことのできる部品として記述して、それらの部品の組み合わせとして全体を記述する、という方法です。

部品の内部が複雑になる場合は、その部品をさらにいくつかの部分に分割して、それぞれの部分を部品にします。つまり、部品を階層的に組み合わせていくことによって全体を構成するということです。そのように階層的に動作を記述したプログラムは、そうでないプログラムよりも、人間にとって理解しやすいものになります。

部品というのは、その内部の構造を知らなくても使うことができるようなものでないといけません。Ruby の場合は、メソッドというのが動作の部品になると考えることができます。つまり、Ruby では、メソッドを階層的に組み合わせたものとして動作を記述することによって、人間にとって理解しやすいプログラムを書くことができるということです。

2.6 条件

2.6.1 真偽値

成り立っているか成り立っていないかを判断することのできる対象のことを、「条件」(condition) と言います。プログラムを書くときには、しばしば、「この数値は 100 よりも大きい」とか、「この文字列は `"wasabi"` と等しい」というような条件が成り立っているかどうかを判断する、という動作を記述することが必要になります。

条件が成り立っているということを「真」(true) と言い、成り立っていないということを「偽」(false) と言います。たとえば、「この数値は 100 よりも大きい」という条件は、「この数値」が 107 のときは真で、94 のときは偽です。

真と偽は、総称して「真偽値」(Boolean value) と呼ばれます。Ruby では、すべてのオブジェクトについて、真偽値のどちらかをあらわしていると解釈することができます。偽をあらわしているのは、`nil` という名前であらわされるオブジェクトと、`false` という名前であらわされるオブジェクトの二つだけです。そして、その二つ以外のすべてのオブジェクトは、真をあらわしていると解釈されます。

ちなみに、`nil` は `NilClass` というクラスのインスタンスで、`false` は `FalseClass` というクラスのインスタンスです。

条件が成り立っているかどうかを調べたいときは、「述語」(predicate) と呼ばれるものを使います。述語というのは、何らかの条件が成り立っているかどうかを調べて、その結果の真偽値をあらわすオブジェクトを返す、という動作をするもののことです（大多数の述語はメソッドですが、メソッドではない述語もあります)。

述語は、普通、真偽値をあらわすオブジェクトとして、`true` または `false` というオブジェクトを返します。`true` という名前であらわされるのは、真をあらわすための専用のオブジェクト

演算子式	説明
a > b	a のほうが b よりも大きい。
a < b	a のほうが b よりも小さい。
a >= b	a のほうが b よりも大きいか、または a と b とは等しい。
a <= b	a のほうが b よりも小さいか、または a と b とは等しい。

表 2.4: 比較演算

で、TrueClass というクラスのインスタンスです。

2.6.2 等値演算

さて、それでは、Ruby の処理系の中に組み込まれている述語のうちで重要なものを紹介していくことにしましょう。まず最初は、== という述語です。

== は、すべてのオブジェクトが持っているメソッドです。このメソッドは、レシーバーと引数とが等しいかどうかを調べて、等しいならば true、そうでなければ false を返します（== という名前は、二項演算子になっています)。たとえば、

```
"karashi" == "karashi"
```

という式を評価すると、true という値が得られ、

```
80 == 27
```

という式を評価すると、false という値が得られます。

二つのオブジェクトのあいだに、等しくないという関係が成り立っているかどうかを調べたいときは、!= という二項演算を使います。たとえば、

```
"karashi" != "karasu"
```

という式を評価すると、true という値が得られ、

```
80 != 80
```

という式を評価すると、false という値が得られます。

なお、== と != は、総称して「等値演算」(equality operation) と呼ばれます。

ちなみに、!= は、独立したメソッドではなくて、== とは逆の動作をするように、== の定義から自動的に作成される演算です。

2.6.3 比較演算

数値と文字列のオブジェクトは、大きいとか小さいという関係が成り立っているかどうかを調べる、「比較演算」(comparison operation) と呼ばれる述語を持っています。比較演算というのは、表 2.4 に示した 4 個の演算のことです（比較演算はすべてメソッドです)。

たとえば、

```
107 > 100
```

という式を評価すると、値として true が得られ、

```
94 > 100
```

という式を評価すると、値として false が得られます。

文字列のオブジェクトが持っている比較演算は、「辞書式順序」(lexicographical order) と呼ばれる順序にもとづいて文字列を比較します。辞書式順序というのは、辞書の見出しを並べるときに使われる順序のことです。文字列の比較演算は、辞書式順序で文字列を並べたときに、前にあるものほど小さく、後ろにあるものほど大きいと判断します。たとえば、

```
"karashi" < "karasu"
```

という式を評価すると、値として true が得られます。

なお、等値演算と比較演算は、総称して「関係演算」(relational operation) と呼ばれます。

演算子式	説明
a `&&` b	a かつ b である。
a `||` b	a または b である。
`!` a	a ではない。

表 2.5: 論理演算

2.6.4 論理演算

二つの条件の組み合わせになっているような条件について判断したり、真偽値を反転させたりしたいときは、「論理演算」(logical operation) と呼ばれる述語を使います。論理演算というのは、表 2.5 に示した 3 個の演算のことで、これらはすべて、真偽値を処理して真偽値を求めるという動作をします（論理演算はメソッドではありません)。

`&&` という論理演算は、左右の式の値が両方とも真のときだけ真を返し、それ以外の場合は偽を返します。つまり、

```
true  &&  true   ⟶  true
true  &&  false  ⟶  false
false &&  true   ⟶  false
false &&  false  ⟶  false
```

という動作をするわけです。

`||` は、左右の式の値が両方とも偽のときだけ偽を返し、それ以外の場合は真を返す論理演算です。したがって、

```
true  ||  true   ⟶  true
true  ||  false  ⟶  true
false ||  true   ⟶  true
false ||  false  ⟶  false
```

という動作をすることになります。

普通、二項演算を呼び出す式を評価すると、二項演算子の左右に書かれた式は、かならず両方とも評価されます。しかし、`&&` または `||` を呼び出す式は、かならずしも両方の式が評価されるとは限りません。左側の式はかならず評価されるのですが、その値だけで結論が出せるならば、右側の式は評価されないのです。`&&` の左側に書かれた式の値が偽だったとすると、それだけで偽という結論を出すことができますので、右側の式は評価されません。同じように、`||` の左側に書かれた式が真だった場合も、右側の式は評価されません。

`!` というのは単項演算で、その右側に書かれた式の値が真ならば偽を返し、偽ならば真を返します。つまり、

```
! true   ⟶  false
! false  ⟶  true
```

というように、真偽値を反転させるという動作をするわけです。

2.6.5 述語の定義

これまでに紹介したように、処理系の中にはさまざまな述語が組み込まれているわけですが、それらの述語は、きわめて単純な条件についての判断しかできません。しかし、述語というのは、プログラムを書く人が自分で定義することも可能ですので、複雑な条件の判断を記述したいときは、その判断をする述語を自分で定義するといいでしょう。

述語を定義するというのは、言い換えれば、戻り値として真偽値を返すメソッドを定義するということです。たとえば、次のプログラムは、レシーバーが偶数であるという条件が成り立っているかどうかを調べる、`even` という述語を定義します。

プログラムの例 `even.rb`

```
class Integer
  def even
```

```
    self % 2 == 0
  end
end
```

実行例

```
irb(main):001:0> load("even.rb")
=> true
irb(main):002:0> 6.even
=> true
irb(main):003:0> 7.even
=> false
```

2.7 選択

2.7.1 選択とは何か

「選択」(selection) という言葉は、日常的に使われるときは、「いくつかのものの中からどれかを選び出すこと」という意味ですが、プログラミングの文脈では、もう少し特殊な意味で使われます。プログラミングの用語としての「選択」は、「いくつかの動作の中からどれかひとつを選び出して実行するという動作」という意味です。

この節では、プログラムの中で選択という動作を記述するためにはどのようなものを書けばいいのか、ということについて説明していきたいと思います。

2.7.2 if 式

まず最初に、真偽値によって動作を選択するという動作を記述する方法について説明することにしましょう。真偽値による動作の選択というのは、条件が成り立っている場合はこの動作を実行して、成り立っていない場合はこの動作を実行する、というような選択のことです。

Ruby では、真偽値による動作の選択を記述したいときは、「if 式」(if expression) と呼ばれる式を書きます。

if 式というのは、

```
if 条件式
   式1
else
   式2
end
```

という形の式のことです。「条件式」のところには、動作を選択するための真偽値を求める式を書きます。そして、条件式の値が真だった場合に評価したい式を「式 $_1$」のところに書いて、条件式の値が偽だった場合に評価したい式を「式 $_2$」のところに書きます。

それでは、if 式を irb に入力してみましょう。まず、条件式の値が真になるような if 式を入力すると、

```
irb(main):001:0> if 5 > 3
irb(main):002:1>   "namako\n".display
irb(main):003:1> else
irb(main):004:1*   "hitode\n".display
irb(main):005:1> end
namako
=> nil
```

というように、else の上に書かれた式が評価されて、namako\n という文字列が出力されます。それに対して、条件式の値が偽になるような if 式を入力すると、

```
irb(main):001:0> if 3 > 5
irb(main):002:1>   "namako\n".display
irb(main):003:1> else
irb(main):004:1*   "hitode\n".display
irb(main):005:1> end
hitode
```

```
   => nil
```

というように、 else の下に書かれた式が評価されて、 hitode\n という文字列が出力されます。

ところで、 if 式というのは式の一種ですから、それを評価すると、その値が得られます。 if 式全体の値は、選択された式の値です。

if 式全体の値についても、irb を使って試してみましょう。そうすると、

```
irb(main):001:0> if 5 == 5
irb(main):002:1>   6006
irb(main):003:1> else
irb(main):004:1*   7117
irb(main):005:1> end
=> 6006
irb(main):006:0> if 5 == 3
irb(main):007:1>   6006
irb(main):008:1> else
irb(main):009:1*   7117
irb(main):010:1> end
=> 7117
```

というように、選択された式の値が if 式全体の値になっている、ということが分かります。

プログラムの例 zero.rb

```
class Numeric
  def zero
    if self == 0
      "zero"
    else
      "not zero"
    end
  end
end
```

実行例

```
irb(main):001:0> load("zero.rb")
=> true
irb(main):002:0> 0.zero
=> "zero"
irb(main):003:0> 5.zero
=> "not zero"
```

プログラムの例 evenodd.rb

```
class Integer
  def even
    self % 2 == 0
  end
  def evenodd
    if even
      "even"
    else
      "odd"
    end
  end
end
```

実行例

```
irb(main):001:0> load("evenodd.rb")
=> true
irb(main):002:0> 6.evenodd
=> "even"
irb(main):003:0> 7.evenodd
=> "odd"
```

2.7.3 else を省略した if 式

真偽値による動作の選択には、条件が真のときは何らかの動作を実行するけれども、条件が偽のときは何もしない、というタイプのものもあります。そのようなタイプの選択を記述したいときは、

```
if 条件式
  式
end
```

というような、else とそれに続く式を省略した形の if 式を書きます。この形の if 式の中に書かれた式は、「条件式」のところに書かれた式の値が真だった場合だけ評価されます。

irb を使って試してみると、次のようになります。

```
irb(main):001:0> if true
irb(main):002:1>   "namako\n".display
irb(main):003:1> end
namako
=> nil
irb(main):004:0> if false
irb(main):005:1>   "namako\n".display
irb(main):006:1> end
=> nil
```

プログラムの例 distime.rb

```
class Integer
  def displayTime
    if self >= 60
      ((self / 60).to_s + "時間").display
    end
    if (self % 60 != 0) || (self == 0)
      ((self % 60).to_s + "分").display
    end
    "\n".display
  end
end
```

実行例

```
irb(main):001:0> load("distime.rb")
=> true
irb(main):002:0> 150.displayTime
2時間30分
=> nil
irb(main):003:0> 180.displayTime
3時間
=> nil
irb(main):004:0> 40.displayTime
40分
=> nil
```

else を省略した形の if 式の値は、その中の式が評価された場合はその値で、評価されなかった場合は nil です。irb を使って試してみると、次のようになります。

```
irb(main):001:0> if true
irb(main):002:1>   1771
irb(main):003:1> end
=> 1771
irb(main):004:0> if false
irb(main):005:1>   1771
irb(main):006:1> end
=> nil
```

2.7.4 elsif を付加した if 式

選択の対象となる動作が 3 個以上あるような選択のことを、「多肢選択」(multibranch selection) と言います。多肢選択には、いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプのものと、ひとつの式の値が何なのかということによって動作を選択するタイプのものがあります。

いくつかの条件のうちのどれが真になるかということによって動作を選択するというタイプの多肢選択は、

```
if 条件式 1
   式 1
elsif 条件式 2
   式 2
  •
  •
  •
elsif 条件式 n
   式 n
else
   式 n+1
end
```

というような、if と else のあいだに elsif を付加した形の if 式を使うことによって記述することができます。

この形の if 式を評価すると、その中の条件式が、値が真になるものが見つかるまで、上から順番に評価されていきます。値が真になる条件式が見つかった場合は、その条件式の下に書かれた式が評価されて、その式の値が if 式全体の値になります。もしも、すべての条件式の値が偽だった場合は、else の下に書かれた式が評価されて、その式の値が if 式全体の値になります。

irb を使って試してみると、次のようになります。

```
irb(main):001:0> if false
irb(main):002:1>   8888
irb(main):003:1> elsif true
irb(main):004:1>   5000
irb(main):005:1> else
irb(main):006:1*   1441
irb(main):007:1> end
=> 5000
```

プログラムの例 sign.rb

```
class Numeric
  def sign
    if self > 0
      "plus"
    elsif self < 0
      "minus"
    else
      "zero"
    end
  end
end
```

実行例

```
irb(main):001:0> load("sign.rb")
=> true
irb(main):002:0> 5.sign
=> "plus"
irb(main):003:0> -5.sign
=> "minus"
irb(main):004:0> 0.sign
```

```
=> "zero"
```

2.7.5 case 式

いくつかの条件のうちのどれが真になるかということではなくて、ひとつの式の値が何なのかということによって動作を選択するというタイプの多肢選択は、if 式を使って記述することもできますが、「case 式」(case expression) と呼ばれる式を使えば、if 式を使うよりもすっきりと記述することができます。

case 式は、

```
case 判定式
when 場合式 1
    式 1
when 場合式 2
    式 2
  •
  •
  •
when 場合式 n
    式 n
else
    式 n+1
end
```

というように書きます。case 式を評価すると、まず最初に、「判定式」というところに書かれた式が評価されます。そして、判定式の値と一致する値を持つ式が見つかるまで、「場合式」というところに書かれた式が、上から順番に評価されていきます。判定式の値と一致する場合式が見つかった場合は、その下に書かれた式が評価されて、その式の値が case 式全体の値になります。判定式の値と一致する場合式が見つからなかった場合は、else の下に書かれた式が評価されて、その式の値が case 式全体の値になります。

irb を使って試してみると、次のようになります。

```
irb(main):001:0> case 2
irb(main):002:1> when 1
irb(main):003:1>   "one"
irb(main):004:1> when 2
irb(main):005:1>   "two"
irb(main):006:1> else
irb(main):007:1*   "many"
irb(main):008:1> end
=> "two"
```

プログラムの例 operate.rb

```
class Numeric
  def operate(ope, a)
    case ope
    when "add"
      self + a
    when "subtract"
      self - a
    when "multiply"
      self * a
    when "divide"
      self / a
    else
      self
    end
  end
end
```

実行例

```
irb(main):001:0> load("operate.rb")
=> true
irb(main):002:0> 23.operate("add", 34)
=> 57
irb(main):003:0> 100.operate("divide", 4)
=> 25
```

なお、case式のwhenの右側には、

when 場合式 , 場合式 , ・・・

というように、2個以上の場合式をコンマで区切って並べることもできます。その場合、whenの右側に書かれた場合式の列は、左から順番に評価されていって、判定式の値と一致するものが見つかったならば、whenの下に書かれた式が評価されます。

2.8 ブロック

2.8.1 ブロックとは何か

メソッドは、引数としていくつかのオブジェクトを受け取ることができるわけですが、メソッドが受け取ることのできるものはオブジェクトだけではありません。メソッドは、「ブロック」(block) と呼ばれるものを受け取ることもできます。

ブロックというのは、包装紙のようなもので式を包み込んだもののことです。この包装紙は、その中に包まれている式を、現在の場所で評価するのではなくて、式という形を保ったままメソッドに送り届ける、という機能を持っています。

ブロックを受け取ったメソッドは、そのブロックの中の式を評価することができます。ブロックの中の式を評価することを、ブロックを「実行する」(execute) と言います。

メソッドがブロックを受け取ることができるということは、メソッドは、オブジェクトだけではなくて、何らかの動作をも処理の対象にすることができる、ということを意味しています。

2.8.2 ブロックの書き方

ブロックの書き方は、二通りあります。ひとつは、doとendという二つの単語を使う書き方です。それらの単語のあいだに式を書くと、その全体がひとつのブロックになります。つまり、

do
　式
end

という形のものを書けば、それがブロックになるということです。

ブロックのもうひとつの書き方は、中括弧 ({}) を使うというものです。式を中括弧 ({}) で囲むと、その全体がひとつのブロックになります。ですから、

{
　式
}

という形のものも、やはりブロックです。

ブロックは、1行にまとめて書いてもかまいません。つまり、ブロックは、

do 式 end

という形で書くこともできるし、

{ 式 }

という形で書くこともできる、ということです。

ブロックは、doとendを使って書いてもかまいませんし、中括弧を使って書いてもかまいません。慣習としては、ブロックを複数行に分けて書くときはdoとend、ブロック全体を1行で書くときは中括弧、という使い分けが定着しているようです。

2.8.3 ブロックを渡すメッセージ

メソッドにブロックを渡したいときは、オブジェクトに送るメッセージの中に、そのブロックを書きます。

メソッドにブロックを渡すメッセージは、

メソッド名 (式 , ···) ブロック

と書きます。たとえば、

```
hibari { tsubame }
```

というメッセージをオブジェクトに送ると、hibari というメソッドが呼び出されて、そのメソッドに対して、

```
{ tsubame }
```

というブロックが渡されます。

2.8.4 ブロックの実行

受け取ったブロックを実行したいときは、「yield 式」(yield expression) と呼ばれる式を書きます。

yield 式は、基本的には、

```
yield
```

と書くだけです。yield 式を評価すると、受け取ったブロックが実行されます。

次のプログラムは、受け取ったブロックを実行するだけ、というきわめて単純な動作をする、exblock というメソッドを定義します。

プログラムの例 exblock.rb

```
def exblock
  yield
end
```

それでは、このメソッドを呼び出して、それに対してブロックを渡してみましょう。

実行例

```
irb(main):001:0> load("exblock.rb")
=> true
irb(main):002:0> exblock { "hototogisu\n".display }
hototogisu
=> nil
```

この exblock のような、受け取ったブロックを実行するように作られているメソッドは、「イテレーター」(iterator) と呼ばれます。iterator という単語は「繰り返すもの」という意味ですが、すべてのイテレーターが何らかの動作を繰り返すとは限りません。

2.8.5 ブロックが受け取る引数

ブロックは、メソッドと同じように、自分が動作を開始する直前に、いくつかのオブジェクトを引数として受け取ることができます。ただし、引数を受け取るためには、あらかじめ、do または左中括弧の直後に、

| 仮引数名 , ··· |

という形のもの、つまり、コンマで区切られた仮引数名の列を縦棒 (|) で囲んだものを書いておく必要があります。そうすると、ブロックに渡された引数が、それらの名前を持つ仮引数に代入されます。たとえば、

```
{ |a, b| a.display; b.display }
```

というブロックは、2 個のオブジェクトを引数として受け取って、それらを出力する、という動作をします。

ブロックに引数を渡して、そのブロックを実行したいときは、

yield(式 , ···)

という形のyield式を書きます。この形のyield式を評価すると、丸括弧の中に書かれた式の値が引数としてブロックに渡されて、ブロックが実行されます。

次のプログラムは、2個の引数とブロックを受け取って、それらの引数を渡してブロックを実行する、exblock2というメソッドを定義します。

プログラムの例 exblock2.rb

```
def exblock2(a, b)
  yield(a, b)
end
```

それでは、このメソッドを呼び出して、それに対して引数とブロックを渡してみましょう。

実行例

```
irb(main):001:0> load("exblock2.rb")
=> true
irb(main):002:0> exblock2(100, 24) { |a, b| (a-b).display }
76=> nil
```

2.8.6 ブロックが返す戻り値

ブロックは、メソッドと同じように、自分の実行が終了したのち、自分を実行したメソッドに対して戻り値を返します。ブロックが返す戻り値というのは、ブロックの中に書かれた式の値です。そして、ブロックが返した戻り値は、そのブロックを実行したyield式の値になります。

それでは、先ほど定義した、exblock2というメソッドを使って、ブロックが本当に戻り値を返すかどうか、試してみましょう。

実行例

```
irb(main):001:0> load("exblock2.rb")
=> true
irb(main):002:0> exblock2(100, 24) { |a, b| a-b }
=> 76
```

2.9 イテレーター

2.9.1 イテレーターと繰り返し

「繰り返し」(iteration) という動作、つまり、ひとつの動作を何回も実行するという動作は、基本的には、イテレーターを使うことによって記述することができます。

イテレーターというのは、前の節で説明したように、受け取ったブロックを実行するメソッドのことですから、すべてのイテレーターが何らかの繰り返しを実行するとは限りません。しかし、大多数のイテレーターは、何らかの繰り返しを実行するように作られています。

この節では、組み込みクラスのオブジェクトが持っているイテレーターのうちの主要なものを紹介したいと思います。

2.9.2 整数のイテレーター

整数のオブジェクトは、times、upto、downtoというイテレーターを持っています。

timesは、レシーバーを回数としてブロックの実行を繰り返すイテレーターです。このイテレーターは、レシーバーをそのまま戻り値として返します。

```
irb(main):001:0> 7.times { "Namako".display }
NamakoNamakoNamakoNamakoNamakoNamakoNamako=> 7
```

timesは、ブロックを実行するたびに、それが何回目の実行なのかということをあらわす整数のオブジェクトを、引数としてブロックに渡します（ただし、最初の実行を0回目と数えます)。

```
irb(main):001:0> 18.times { |i| i.display; " ".display }
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 => 18
```

次のプログラムの中で定義されているpowerというメソッドは、整数nを引数として受け取って、レシーバーの0乗から$n-1$乗までを出力します。

プログラムの例 power.rb

```
class Numeric
  def power(n)
    n.times do |i|
      (to_s + "の" + i.to_s + "乗: " +
      (self**i).to_s + "\n").display
    end
  end
end
```

実行例

```
irb(main):001:0> load("power.rb")
=> true
irb(main):002:0> 7.power(10)
7 の 0 乗: 1
7 の 1 乗: 7
7 の 2 乗: 49
7 の 3 乗: 343
7 の 4 乗: 2401
7 の 5 乗: 16807
7 の 6 乗: 117649
7 の 7 乗: 823543
7 の 8 乗: 5764801
7 の 9 乗: 40353607
=> 10
```

upto も、 times と同じように、ブロックに整数を渡してブロックの実行を繰り返すイテレーターです。ブロックに渡す引数は、レシーバーから出発して、1 ずつ増やしていきます。

n と m が整数だとするとき、

n.upto(m) ブロック

という式で upto を呼び出すと、 upto は、

n、$n+1$、$n+2$、・・・、m

という整数のそれぞれをブロックに渡して、ブロックの実行を繰り返します。そして、レシーバーをそのまま戻り値として返します。

```
irb(main):001:0> 20.upto(30) { |i| i.display; " ".display }
20 21 22 23 24 25 26 27 28 29 30 => 20
```

次のプログラムの中で定義されている divisor というメソッドは、レシーバーのすべての約数 (divisor) を出力します。

プログラムの例 divisor.rb

```
class Integer
  def divisor
    (to_s + "の約数: ").display
    1.upto(self) do |i|
      if self % i == 0
        (i.to_s + " ").display
      end
    end
    "\n".display
  end
end
```

実行例

```
irb(main):001:0> load("divisor.rb")
=> true
irb(main):002:0> 36.divisor
36 の約数: 1 2 3 4 6 9 12 18 36
=> nil
```

次のプログラムの中で定義されている eachDivisor というイテレーターは、レシーバーの約

数のそれぞれをブロックに渡してブロックを実行する、ということを繰り返します。そして、sumOfDivisor というメソッドは、eachDivisor を使ってレシーバーのすべての約数の和を求めて、その結果を戻り値として返します。

プログラムの例 sumdiv.rb

```
class Integer
  def eachDivisor
    1.upto(self) do |i|
      if self % i == 0
        yield(i)
      end
    end
    self
  end
  def sumOfDivisor
    sum = 0
    eachDivisor do |i|
      sum += i
    end
    sum
  end
end
```

実行例

```
irb(main):001:0> load("sumdiv.rb")
=> true
irb(main):002:0> 54.sumOfDivisor
=> 120
```

downto は、upto とは逆の動作をするイテレーターです。レシーバーから出発するという点は upto と同じですが、増やすのではなくて、1 ずつ減らしながらブロックの実行を繰り返します。

n と m が整数だとするとき、

n.downto(m) ブロック

という式で downto を呼び出すと、downto は、

n、$n-1$、$n-2$、・・・、m

という整数のそれぞれをブロックに渡して、ブロックの実行を繰り返します。そして、レシーバーをそのまま戻り値として返します。

```
irb(main):001:0> 30.downto(20) { |i| i.display; " ".display }
30 29 28 27 26 25 24 23 22 21 20 => 30
```

2.9.3 数値のイテレーター

整数と浮動小数点数の総称として、それらを「数値」と呼ぶことにしましょう。

数値のオブジェクトは、step というイテレーターを持っています。

step は、数値をブロックに渡してブロックの実行を繰り返すイテレーターです。ブロックに渡す数値は、レシーバーから出発して、一定の歩幅で増加または減少していきます。

b と e と s が数値だとするとき、

b.step(e, s) ブロック

という式で step を呼び出すと、step は、

b、$b+s$、$b+2s$、$b+3s$、・・・

という数値のそれぞれをブロックに渡して、ブロックの実行を繰り返します（s がプラスの場合は増加していって、マイナスの場合は減少していくことになります）。そして、e を通り過ぎたならば繰り返しを終了します。戻り値はレシーバーです。

```
irb(main):001:0> 0.step(80, 7) { |i| i.display; " ".display }
0 7 14 21 28 35 42 49 56 63 70 77 => 0
irb(main):002:0> 80.step(0, -7) { |i| i.display; " ".display }
```

```
80 73 66 59 52 45 38 31 24 17 10 3 => 80
```

2.9.4 範囲のイテレーター

範囲のオブジェクトは、eachというイテレーターを持っています。

eachは、レシーバーに含まれているそれぞれのオブジェクトをブロックに渡して、ブロックの実行を繰り返すイテレーターです。このイテレーターは、レシーバーをそのまま戻り値として返します。

```
irb(main):001:0> ("a".."z").each { |s| s.display }
abcdefghijklmnopqrstuvwxyz=> "a".."z"
```

2.9.5 文字列のイテレーター

文字列のオブジェクトは、each_byteというイテレーターを持っています。

each_byteは、レシーバーを構成しているそれぞれの文字（文字コード）をブロックに渡して、ブロックの実行を繰り返すイテレーターです。このイテレーターは、レシーバーをそのまま戻り値として返します。

たとえば、

```
"Umberto Eco".each_byte { |c| c.display; " ".display }
```

というメッセージ式を評価すると、

```
85 109 98 101 114 116 111 32 69 99 111
```

というように、レシーバーを構成するそれぞれの文字の文字コードが出力されます。

文字コードを文字列に変換したいときは、整数が持っているchrというメソッドを使います。chrは、レシーバーを文字コードとする文字から構成される、長さが1の文字列を返すメソッドです。たとえば、スラッシュ(/)という文字の文字コードは47ですから、

```
47.chr
```

という式を評価すると、"/"という文字列が値として得られます。

次のプログラムの中で定義されているabbreviateというメソッドは、レシーバーの中に含まれている英字の大文字だけを取り出して並べた文字列を戻り値として返します。

プログラムの例 abbrevi.rb

```
class Fixnum
  def isUppercase
    self >= ?A && self <= ?Z
  end
end

class String
  def abbreviate
    abbr = ""
    each_byte do |c|
      if c.isUppercase
        abbr += c.chr
      end
    end
    abbr
  end
end
```

実行例

```
irb(main):001:0> load("abbrevi.rb")
=> true
irb(main):002:0> "Unidentified Flying Object".abbreviate
=> "UFO"
```

次のプログラムの中で定義されているeachCharというイテレーターは、レシーバーを構成しているそれぞれの文字の文字コードをブロックに渡してブロックを実行する、ということを繰り返します。そして、ブロックが返した戻り値を連結することによってできる文字列を戻り値とし

て返します。そして、enclose というメソッドは、eachChar を使って、レシーバーを構成するそれぞれの文字を角括弧で囲むことによってできる文字列を戻り値として返します。

プログラムの例 enclose.rb

```
class String
  def eachChar
    s = ""
    each_byte do |c|
      s += yield(c)
    end
    s
  end
  def enclose
    eachChar do |c|
      "[" + c.chr + "]"
    end
  end
end
```

実行例

```
irb(main):001:0> load("enclose.rb")
=> true
irb(main):002:0> "Italo Calvino".enclose
=> "[I][t][a][l][o][ ][C][a][l][v][i][n][o]"
```

第 1.4 節で紹介した gsub というメソッドを、ここで再び紹介したいと思います。

gsub は、文字列のオブジェクトが持っているメソッドです。このメソッドは、引数として 2 個の文字列を受け取って、1 個目の引数で指定された部分文字列をレシーバーの中で探索して、発見したすべての部分文字列を 2 個目の引数に置き換えることによってできる文字列を戻り値として返します。

実は、この gsub というメソッドは、イテレーターです。つまり、受け取ったブロックを実行するメソッドなのです。イテレーターとしての gsub は、発見した部分文字列を、ブロックを使って別の文字列に変換します。

s と t が文字列だとするとき、

s.gsub(t) ブロック

という式で gsub を呼び出すと、gsub は、s の先頭から末尾に向かって、t を部分文字列として探索します。そして、部分文字列を発見するたびに、その部分文字列をブロックに渡して、ブロックを実行します。そして、s の中で発見されたすべての部分文字列をブロックの戻り値で置き換えることによってできる文字列を戻り値として返します。

次のプログラムの中で定義されている gsubnum というメソッドは、1 個の文字列を引数として受け取って、それがレシーバーの中に部分文字列として含まれている部分に番号を挿入して、その番号と部分文字列とを角括弧で囲むことによってできた文字列を戻り値として返します。

プログラムの例 gsubnum.rb

```
class String
  def gsubnum(s)
    num = 0
    gsub(s) do |ss|
      num += 1
      "[" + num.to_s + ss + "]"
    end
  end
end
```

実行例

```
irb(main):001:0> load("gsubnum.rb")
=> true
irb(main):002:0> "namamuginamagomenamatamago".gsubnum("ma")
=> "na[1ma]mugina[2ma]gomena[3ma]ta[4ma]go"
```

2.10 条件による繰り返し

2.10.1 条件による繰り返しとは何か

繰り返しを記述したいときは、基本的にはイテレーターを使えばいいわけですが、どのような繰り返しの場合でも、イテレーターを使うことが最善の選択肢だとは限りません。イテレーター以外の方法で記述するほうが自然な繰り返し、というものも存在します。それは、条件による繰り返しです。

条件による繰り返しというのは、動作を実行するたびに、その繰り返しを続行するか、それとも繰り返しを終了するか、ということを、何らかの条件が成り立っているかどうかを判断することによって決定する、というタイプの繰り返しのことです。このようなタイプの繰り返しは、イテレーターではなく、繰り返しのための式を使って記述するほうが自然です。

2.10.2 `while` 式

繰り返しのための式にはいくつかの種類があるのですが、それらの中でもっとも基本的なのは、「`while` 式」(`while` expression) と呼ばれる式です。

`while` 式は、

```
while 条件式
  式
end
```

という構文を持つ式です。繰り返しを続行するための条件をあらわす式を「条件式」のところに書いて、繰り返しの対象となる動作をあらわす式をその下に書きます。

`while` 式を評価すると、条件式の評価と、繰り返しの対象となる式の評価とが繰り返されていきます。そして、条件式を評価したときにその値が偽だった場合、`while` 式の評価はそこで終了します。

`while` 式では、条件式が評価されたのちに繰り返しの対象となる式が評価されますので、条件式の値が最初から偽だった場合は、繰り返しの対象となる式は 1 回も実行されないで終了することになります。

2.10.3 条件による繰り返しの例

条件による繰り返しの例として、二つの整数の最大公約数 (greatest common measure) を求めるという処理について考えてみることにしましょう。

二つの整数の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる方法を使えば、きわめて簡単に求めることができます。ユークリッドの互除法というのは、

ステップ 1　与えられた二つの整数のそれぞれを、n と m という変数に代入する。
ステップ 2　m が 0 ならば計算を終了する。
ステップ 3　n を m で除算して、そのあまりを r という変数に代入する。
ステップ 4　m を n に代入する。
ステップ 5　r を m に代入する。
ステップ 6　ステップ 2 に戻る。

という計算を実行していけば、計算が終了したときの n が、最初に与えられた二つの整数の最大公約数になっている、というものです。ステップ 2 からステップ 6 までは、

　m が 0 ではないあいだ、ステップ 3 からステップ 5 までを繰り返す。

ということだと考えることができますので、その部分は、`while` 式を書くことによって素直に記述することができます。

さて、それでは、レシーバーと引数との最大公約数を求めて、その結果を戻り値として返す、`gcm` というメソッドの定義を書いてみましょう。

プログラムの例　`gcm.rb`

```
class Integer
  def gcm(m)
    n = self
    while m != 0
```

```
      r = n % m
      n = m
      m = r
    end
    n
  end
end
```

実行例

```
irb(main):001:0> load("gcm.rb")
=> true
irb(main):002:0> 54.gcm(36)
=> 18
```

2.11 再帰

2.11.1 再帰的な構造

全体と同じ構造のものが一部分として含まれている構造のことを、「再帰的な」(recursive) 構造と言います。

雑誌の表紙に、ときおり、同じ雑誌の同じ号を手に持った人物が登場することがあります。その人物が持っている雑誌の表紙には、同じ人物がいて、同じ雑誌を持っています。すると、表紙の中に同じ表紙がある、という構造が無限に続くことになります。このような構造は、再帰的な構造の一例です。

抽象的な概念のうちにも、再帰的な構造を持つものがけっこうあります。たとえば、これから説明する「括弧列」というのもそのひとつです。

n が 0 またはプラスの整数だとするとき、n 個の左丸括弧の列の右側に n 個の右丸括弧の列を並べることによってできる文字列のことを、n 重の「括弧列」と呼ぶことにします。たとえば、

```
(((((((())))))))
```

という文字列は、7 重の括弧列です。

括弧列が持っている再帰的な構造というのは、丸括弧で囲まれた内側にさらに括弧列があるという構造のことです。つまり、n が 1 以上のとき、n 重の括弧列は、

($\boxed{(n-1)\text{ 重の括弧列}}$)

という文字列だと考えることができる、ということです。このように、括弧列は、その一部分として全体と同じ構造のものを含んでいるわけです。

2.11.2 再帰的なメソッド

メソッドは、再帰的に定義することが可能です。メソッドを再帰的に定義するというのは、自分自身というメソッドを使ってメソッドを定義するということです。再帰的な構造を持った概念を取り扱うメソッドは、再帰的に定義するほうが、再帰的ではない方法で定義するよりもすっきりした記述になります。

それでは、先ほど説明した括弧列という再帰的な概念を使って、メソッドの再帰的な定義について考えてみることにしましょう。レシーバーを n とする括弧列を作る paren というメソッドを再帰的に定義するためにはどうすればいいでしょうか。

メソッドを再帰的に定義するためには、まず、「基底」と呼ばれるものについて考える必要があります。

「基底」(basis) というのは、再帰的な構造の中心にあって、それ以上の内部を持っていないもののことです。たとえば、括弧列では、もっとも内側の丸括弧のさらに内側にある空文字列（長さが 0 の文字列）が基底です。

再帰的なメソッドは、かならずひとつの選択を持っています。それは、再帰が必要な場合とそうでない場合との選択です。再帰が必要ではない場合というのは、基底を取り扱う場合のことです。

括弧列を作る paren の場合は、n が 1 以上かどうかで動作を選択します。n が 1 以上ならば、自分自身を呼び出して $(n-1)$ 重の括弧列を求めて、それをさらに丸括弧で囲んだものを返します。n が 0 ならば、基底、すなわち空文字列を返します。

次のプログラムは、以上の考え方に基づいて paren を定義したものです。

プログラムの例 paren.rb

```
class Integer
  def paren
    if self >= 1
      "(" + (self - 1).paren + ")"
    else
      ""
    end
  end
end
```

実行例

```
irb(main):001:0> load("paren.rb")
=> true
irb(main):002:0> 14.paren
=> "((((((((((((((()))))))))))))))"
```

2.11.3 階乗

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて掛け算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果のことを、n の「階乗」(factorial) と呼んで、$n!$ と書きあらわします。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

この階乗というのも、再帰的な構造を持つ概念の一例です。なぜなら、$n!$ を求める計算は、

$$\begin{cases} 0! = 1 \\ n \geq 1 \text{ ならば } \quad n! = n \times (n-1)! \end{cases}$$

ということだと考えることができるからです。

ですから、階乗を求めるメソッドは、自分自身を使うことによって、かなりすっきりと定義することができます。次のプログラムは、階乗を求める factorial というメソッドを再帰的に定義したものです。

プログラムの例 fact.rb

```
class Integer
  def factorial
    if self >= 1
      self * (self - 1).factorial
    else
      1
    end
  end
end
```

実行例

```
irb(main):001:0> load("fact.rb")
=> true
irb(main):002:0> 30.factorial
=> 265252859812191058636308480000000
```

2.11.4 フィボナッチ数列

フィボナッチ数列 (Fibonacci sequence) と呼ばれる数列の第 n 項を求める計算も、再帰的な構造を持っています。

フィボナッチ数列の第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

という計算によって求めることができます。たとえば、第 7 項を求めたいときは、第 5 項と第 6 項とを加算すればいいわけです。

次のプログラムは、フィボナッチ数列の第 n 項を求める fibonacci というメソッドを再帰的に定義したものです。

プログラムの例　fibona.rb

```
class Integer
  def fibonacci
    if self >= 2
      (self - 2).fibonacci + (self - 1).fibonacci
    else
      1
    end
  end
end
```

実行例

```
irb(main):001:0> load("fibona.rb")
=> true
irb(main):002:0> 30.fibonacci
=> 1346269
```

第 3 章　クラス

3.1　クラスの定義

3.1.1　クラスを定義する必要性

Ruby では、数値や文字列のような、多くのプログラムで必要とされる基本的なオブジェクトは、組み込みクラス、つまり処理系の中に最初から組み込まれているクラスから生成されます。しかし、プログラムの中で必要となるオブジェクトは、かならずしも組み込みクラスのインスタンスだけとは限りません。特殊な問題を扱うプログラムは、その問題に即した特殊なオブジェクトを必要とします。たとえば、図形を扱うプログラムは点や線をあらわすオブジェクトを必要とするでしょうし、ロールプレイングゲームのプログラムはキャラクターやマップやアイテムをあらわすオブジェクトを必要とするでしょう。

組み込みクラスのインスタンスではない独自のオブジェクトを使うためには、それを生成する新しいクラスを作る必要があります。クラスを作ることを、クラスを「定義する」(define) と言います。

3.1.2　クラスを定義する方法

クラスを定義したいときは、まだ存在していないクラスの名前を指定したクラス定義を書きます。

クラス定義というのは、第 2.5 節で説明したように、

```
class クラス名
  式
end
```

という形の式のことです。「クラス名」のところに既存のクラスの名前を書くと、既存のクラスにメソッドの鋳型を追加することになるわけですが、既存のクラスの名前ではなくて、まだ存在していないクラスの名前を書くと、その名前を持つ新しいクラスが定義されることになります。

クラスの名前は、変数の名前と同じように、英字、数字、アンダースコア (_) を並べることに

よって作ります。ただし、先頭の文字は、英字の大文字でないといけません。たとえば、Namako、Uni587、Ika_tako、IkaTakoなどは、クラスに与えることのできる名前です。

それでは、実際にクラスを定義してみましょう。irbに、

```
class Hitode
end
```

という式を入力してみてください。すると、Hitodeという名前の新しいクラスが定義されます。次に、新しいクラスが実際にできたかどうかを確かめるために、それにclassというメッセージを送ってみましょう。クラスというのは、Classというクラスのインスタンスですから、irbに、

```
Hitode.class
```

という式を入力すると、戻り値としてClassというクラスが得られるはずです。

クラス定義の中にメソッド定義を書いておくと、そのメソッド定義によって定義されたメソッドの鋳型がクラスに追加されます。したがって、そのクラスから生成されたすべてのインスタンスは、そのメソッドの鋳型から生成されたメソッドを持つことになります。

3.1.3 インスタンスの生成

クラス定義を書くことによって定義されたクラスからインスタンスを生成したいときは、いったいどうすればいいのでしょうか。

まず、次のプログラムを入力して、ファイルに保存してください。

プログラムの例 kurage.rb

```
class Kurage
  def what
    "kurage"
  end
end
```

次に、このプログラムをirbで実行してください。すると、Kurageという名前のクラスが定義されて、そのクラスの中に、whatという名前のメソッドを生成する鋳型が追加されるはずです。

それでは次に、このKurageというクラスからインスタンスを生成してみましょう。

クラス定義によって新しく定義されたクラスは、かならず、newという名前のメソッドを持っています。newは、レシーバーのインスタンスを生成して、そのインスタンスを戻り値として返す、という動作をします。ですから、クラス定義によって定義されたクラスのインスタンスを生成したいときは、そのクラスにnewというメッセージを送ればいいわけです。

それでは、Kurageクラスにnewを送ってみましょう。irbに、

```
k = Kurage.new
```

という式を入力してください。すると、Kurageクラスのインスタンスが生成されて、それがkという変数に代入されます。ですから、

```
k.what
```

という式で、変数kが指し示しているオブジェクトにwhatというメッセージを送ると、kurageという文字列が値として得られるはずです。

3.1.4 インスタンス変数

クラスというのがオブジェクトの種類をあらわしているのに対して、オブジェクトというのは個々のものをあらわしています。個々のものがいくつかあるとするとき、たとえそれらが同じ種類のものだとしても、それぞれのものはどこかが違っているのが普通です。つまり、個々のものは何らかの個性を持っているということです。

ところで、先ほど定義したKurageクラスからいくつかのインスタンスを生成したとするとき、それらのインスタンスは個性を持っていると言えるでしょうか。答はノーです。その理由は、それらのインスタンスは、個性を表現するものをその内部に持っていないからです[1]。

オブジェクトは、普通、変数とメソッドから構成されます。オブジェクトの中の変数は、個々のオブジェクトの個性を表現しているオブジェクトを指し示すために使われます。

[1]ただし、それぞれのインスタンスは、自分を識別することのできる番号を持っています。

変数については第 2.4 節で説明しましたが、そこで説明した変数というのは、「ローカル変数」(local variable) と呼ばれる種類のものでした。

変数を取り扱うことのできる、時間的、空間的、または記述の上での範囲のことを、その変数の「スコープ」(scope) と呼びます。

ローカル変数というのは、ひとつのメソッドの中の限られた部分が実行されているあいだだけ存在する変数です。そのようなスコープを持っているため、ローカル変数は、オブジェクトの個性を表現しているオブジェクトを指し示すという目的には使うことができません。オブジェクトの個性を表現しているオブジェクトを指し示すためには、ひとつのメソッドが実行されているあいだだけではなくて、ひとつのオブジェクトが存在している限り存在し続ける、というスコープを持つ変数を使う必要があります。

ひとつのオブジェクトが存在している限り存在し続ける変数は、「インスタンス変数」(instance variable) と呼ばれます。変数に付ける名前の先頭の文字をアットマーク (`@`) にすると、その変数はインスタンス変数になります。

それでは、次のプログラムを入力して、ファイルに保存してください。

プログラムの例 `ningen.rb`

```
class Ningen
  def setNamae(namae)
    @namae = namae
  end
  def getNamae
    @namae
  end
end
```

このプログラムの中で定義されている `Ningen` というクラスから生成されたインスタンスは、`@namae` というインスタンス変数を持っています。

それでは、`Ningen` クラスのインスタンスに個性を与えてみましょう。まず最初に、

```
n1 = Ningen.new
n2 = Ningen.new
```

という二つの式で、`Ningen` クラスの二つのインスタンスを生成して、それらを `n1` と `n2` に代入してください。次に、

```
n1.setNamae("Umino Miyuki")
n2.setNamae("Moroboshi Ataru")
```

という二つの式で、それぞれのインスタンスが持っている `setNamae` というメソッドを呼び出してください。そうすると、それぞれのインスタンスの中にある `@namae` というインスタンス変数に、異なる文字列が代入されます。ですから、

```
n1.getNamae
n2.getNamae
```

という二つの式で、それぞれのインスタンスに `getNamae` というメッセージを送ると、それぞれのインスタンスの `getNamae` は、異なる文字列を返すことになります。

3.1.5 カプセル化

Ruby では、オブジェクトの外部にあるメソッドはインスタンス変数にアクセスすることができない、という仕様になっています。そのような仕様にしている目的は、「カプセル化」(encapsulation) と呼ばれるものを実現するためです。さてそれでは、「カプセル化」というのはいったいどういうことなのでしょうか。

オブジェクトというのは、プログラムを構成する部品の一種です。部品というものを設計する上で重要なことは、部品を実現するメカニズムと、部品の使い方とを分離するということです。メカニズムと使い方とが互いに依存していると、必要に応じてメカニズムを変更しなければならなくなったときに、使い方まで変更しなければならなくなってしまって、不便だからです。

部品の設計者がメカニズムと使い方とを分離したいと思っても、部品の外部からメカニズムが丸見えになっていれば、メカニズムに依存した使い方が可能になってしまいます。ですから、部品自体に、内部のメカニズムを外側から扱うことができないようにする機能が必要になります。

そのような機能を使って、使ってもかまわない部分だけを公開して、それ以外の部分を隠蔽することを、部品を「カプセル化する」(encapsulate) と言います。「カプセル化」(encapsulation) は、その名詞形です。

オブジェクトの内部にあるインスタンス変数は、普通、そのオブジェクトを実現するためのメカニズムとのあいだに密接な関係があります。ですから、オブジェクトの外部にあるメソッドはインスタンス変数にアクセスすることができないという Ruby の仕様は、オブジェクトをカプセル化するという目的に大いに貢献することになります。

3.1.6 インスタンス変数の初期化

新しく作られた変数に最初に代入されるオブジェクトのことを、その変数の「初期値」(initial value) と呼びます。そして、変数に初期値を代入することを、変数を「初期化する」(initialize) と言います。

クラスからインスタンスを生成したときに、そのインスタンスの中にあるインスタンス変数をその時点で初期化したい、ということがしばしばあります。そのような初期化は、`initialize` という名前のメソッドを定義することによって実現することができます。

`initialize` という名前のメソッドを定義すると、そのメソッドは、クラスからインスタンスが生成された時点で自動的に呼び出されます。ですから、インスタンス変数に初期値を代入するように `initialize` を定義しておけば、インスタンス変数は、インスタンスが生成された時点で初期化されることになります。

それでは、次のプログラムを入力して、ファイルに保存してください。

プログラムの例 `saifu.rb`

```
class Saifu
  def initialize
    @kingaku = 1000
  end
  def dashiire(kingaku)
    @kingaku += kingaku
  end
end
```

このプログラムの中で定義されている `Saifu` というクラスからインスタンスを生成すると、その時点で `initialize` が呼び出されて、`@kingaku` という変数に 1000 が代入されます。ですから、

```
s = Saifu.new
```

という式で、`Saifu` クラスのインスタンスを `s` という変数に代入したのち、

```
s.dashiire(0)
```

という式で、そのインスタンスに `dashiire(0)` というメッセージを送ると、その結果として 1000 という整数が得られます。

3.1.7 `new` の引数

クラスが持っている `new` というメソッドは、引数を受け取ることができます。`new` に引数を渡すと、`new` は、自分が生成したインスタンスの中にある `initialize` に、それらの引数をそのまま渡します。

ですから、引数を受け取るように `initialize` を定義しておくことによって、インスタンス変数を初期化するためのオブジェクトを、インスタンスを生成するときに指定することができるようになります。

次のプログラムは、先ほどのプログラムを、`@kingaku` の初期値を `new` に引数として渡すことができるように改良したものです。

プログラムの例 `saifu2.rb`

```
class Saifu
  def initialize(kingaku)
    @kingaku = kingaku
  end
  def dashiire(kingaku)
    @kingaku += kingaku
```

```
  end
end
```

この改良版の initialize は、1 個の引数を受け取って、それを @kingaku というインスタンス変数に初期値として代入します。ですから、

```
s = Saifu.new(3700)
```

というように、Saifu クラスからインスタンスを生成するときに new に引数を渡せば、new がその引数を initialize に渡しますので、その引数が @kingaku に初期値として代入されることになります。

3.1.8 クラスメソッド

クラスというのもオブジェクトの一種ですので、クラスは、メソッドの鋳型だけではなくて、メソッドそのものを持つことも可能です。クラスが持っているメソッドは、「クラスメソッド」(class method) と呼ばれます。たとえば、インスタンスを生成する new や、スーパークラスを求める superclass は、クラスメソッドの例です。

クラス定義の中にメソッド定義を書くと、普通は、メソッドではなくてメソッドを生成する鋳型がクラスに追加されます。さて、それでは、メソッドの鋳型ではなくて、メソッドそのものをクラスに追加したいときは、いったいどうすればいいのでしょうか。

メソッド定義を書くとき、メソッド名の左側に、クラス名とドット (.) を付けると、クラスに追加されるのは、そのメソッドを生成する鋳型ではなくて、メソッドそのものになります。つまり、

```
def クラス名.メソッド名(仮引数名, ··· )
  式
end
```

という形のメソッド定義を書くことによって、クラスメソッドを定義することができるわけです。

それでは、クラスメソッドを実際に定義してみましょう。まず、次のプログラムを入力してください。

プログラムの例 namako.rb

```
class Namako
  def what
    "namako"
  end
  def Namako.what
    "class"
  end
end
```

このプログラムの中では、what という同じ名前を持つメソッドが二つ定義されていますが、ひとつは Namako クラスのインスタンスの中に作られるメソッドで、もうひとつは Namako クラスの中に作られるメソッドです。

それでは、このプログラムを load で実行したのち、

```
Namako.new.what
```

という式で、Namako クラスのインスタンスに what というメッセージを送ってみてください。すると、そのインスタンスが持っている what が呼び出されますので、namako という文字列が得られるはずです。次に、

```
Namako.what
```

という式で、Namako というクラスに what というメッセージを送ってみてください。すると、クラスメソッドの what が呼び出されますので、class という文字列が得られるはずです。

3.2 サブクラスの定義

3.2.1 スーパークラスを指定しないクラス定義

Ruby では、Object 以外のすべてのクラスは、かならず、自分のスーパークラスを持っています。新しいクラスを定義した場合も、その新しいクラスは、何らかの既存のクラスをスーパークラスとして持つことになります。

前の節で、新しいクラスを定義するためのクラス定義をいくつか書きましたが、それらのクラス定義の中には、新しく作るクラスのスーパークラスが何なのかということを指定する記述はありませんでした。そのように、スーパークラスを指定しないで新しいクラスを定義した場合は、スーパークラスとして Object が指定されたと解釈されます。たとえば、

```
class Kingyo
end
```

という、スーパークラスを指定する記述を含んでいないクラス定義で、Kingyo というクラスを定義したとしましょう。この場合、Kingyo のスーパークラスは Object ですから、

```
Kingyo.superclass
```

という式で、Kingyo に superclass というメッセージを送ると、Object が値として得られます。

3.2.2 スーパークラスを指定したクラス定義

さて、それでは、Object 以外のクラスをスーパークラスとして指定して新しいクラスを定義するためには、クラス定義の中にどのような記述を書けばいいのでしょうか。

クラス定義を書くときに、定義されるクラスの名前の右側に小なり (<) を書いて、すでに存在するクラスの名前をそのさらに右側に書くと、定義されるクラスは、小なりの右側に書かれたクラスのサブクラスになります。つまり、

```
class クラス名 < スーパークラス名
  式
end
```

という形のクラス定義を書くことによって、スーパークラスを指定して新しいクラスを定義することができるということです。

それでは、次の二つのクラス定義を irb に入力してください。

```
class Sakana
end
class Unagi < Sakana
end
```

これらのクラス定義によって定義される Sakana と Unagi という二つのクラスのあいだには、Sakana は Unagi のスーパークラスで、Unagi は Sakana のサブクラスだ、という関係が生じます。ですから、

```
Unagi.superclass
```

という式を評価すると、Sakana という値が得られます。

3.2.3 継承

オブジェクト指向をサポートするプログラミング言語の大多数は、スーパークラスが持っている変数やメソッドの鋳型がサブクラスに自動的に受け継がれるという機能を持っていて、そのような機能は「継承」(inheritance) と呼ばれます。そして、継承によって変数やメソッドの鋳型を受け継ぐことを、変数やメソッドの鋳型を「継承する」(inherit) と言います。

Ruby も、継承という機能を備えているプログラミング言語のひとつです。それでは、irb を使って、本当に変数やメソッドの鋳型が継承されるかどうかを確かめてみましょう。まず、次のプログラムを入力してください。

プログラムの例 rikishi.rb

```
class Ningen
  def setNamae(namae)
    @namae = namae
  end
```

```
  def getNamae
    @namae
  end
end

class Rikishi < Ningen
  def setShikona(shikona)
    @shikona = shikona
  end
  def getShikona
    @shikona
  end
end
```

このプログラムの中で定義されているNingenは、人間をあらわすオブジェクトを生成するクラスです。そして、Rikishiは、Ningenクラスのサブクラスで、力士をあらわすオブジェクトを生成します。

次に、このプログラムをloadで実行したのち、

```
r = Rikishi.new
```

という式で、Rikishiクラスのインスタンスを生成して、それをrという変数に代入してください。

Rikishiクラスは Ningenクラスのサブクラスですから、Rikishiクラスのインスタンスは、@namaeというインスタンス変数と、setNamae、getNamaeというメソッドを持っているはずです。

それでは、実際にメソッドを呼び出してみましょう。まず、

```
r.setNamae("Tamura Yoshio")
```

という式を入力してください。すると、rが指し示しているオブジェクトに対して名前が設定されます。そして次に、

```
r.getNamae
```

という式を入力してください。すると、設定されている名前が値として得られます。

Rikishiクラスのインスタンスは、Ningenクラスから継承した変数とメソッドだけではなくて、力士の四股名を指し示す@shikonaという独自のインスタンス変数と、setShikona、getShikonaという独自のメソッドも持っています。ですから、

```
r.setShikona("Tokinoumi")
```

という式で、rが指し示しているオブジェクトに四股名を設定したのち、

```
r.getShikona
```

という式を評価すると、設定されている四股名が値として得られるはずです。

3.2.4 継承を利用することの意味

スーパークラスとサブクラスの関係というのは、分類項目の大分類と小分類の関係だと考えることができます。つまり、スーパークラスというのはサブクラスよりも一般的な種類をあらわしていて、サブクラスというのはスーパークラスよりも特殊な種類をあらわしている、ということです。ですから、スーパークラスを指定して新しいクラスを作るということは、すでに存在する何らかのクラスを特殊化することによって新しいクラスを作るということを意味しています。

特殊なクラスは、それよりも一般的なクラスが持っている性質をすべて持っていて、さらに独自の性質も持っている必要があります。クラスの性質というのは、具体的には、そのインスタンスが持っている変数やメソッドのことだと考えていいでしょう。つまり、継承という機能は、スーパークラスはサブクラスよりも一般的で、サブクラスはスーパークラスよりも特殊なものだという関係を実現するために存在しているわけです。

プログラムを書くとき、継承という機能を上手に使うことは、さまざまなメリットを与えてくれます。主要なメリットとしては、次のようなものがあります。

(1) プログラムの読みやすさを向上させることができます。いくつかのクラスから構成されるプログラムを書く場合、それらのクラスが互いに無関係に存在しているよりも、一般的なもの

と特殊なものという関係で木の形に整理されているほうが、はるかに読みやすいプログラムになります。

(2) プログラムを書くという作業の効率を向上させることができます。たとえば、A と B という互いによく似たクラスを作る場合、A と B をまったく別々のクラスとして定義するよりも、それらに共通する性質を持つクラスをまず定義して、そののち、A と B をそのサブクラスとして定義するほうが、効率的です。

(3) 再利用しやすいプログラムを書くことができます。継承を使って書かれたプログラムの中には、目的に応じた特殊な機能だけではなくて、それを一般化した機能が、スーパークラスの定義という形で記述されています。そのような一般的な機能を記述した部分は、特殊な機能を記述した部分よりも、ほかのプログラムを書くときに再利用しやすくなっているのです。

3.2.5 オーバーライド

新しいクラスを定義するとき、新しいインスタンス変数やメソッドを追加するだけではなくて、スーパークラスで定義されているメソッドの動作を変更したい、ということがしばしばあります。スーパークラスのメソッドの動作をサブクラスで変更したいときは、スーパークラスで定義されているメソッドと同じ名前のメソッドをサブクラスで定義します。このことを、スーパークラスのメソッドを「オーバーライドする」(override) と言います。

それでは、実際にスーパークラスのメソッドをオーバーライドしてみましょう。まず、次のプログラムを入力して、 load で実行してください。

プログラムの例 maguro.rb

```
class Sakana
  def what
    "sakana"
  end
end

class Maguro < Sakana
  def what
    "maguro"
  end
end
```

次に、 Sakana クラスのインスタンスが持っている what というメソッドを呼び出す、

```
Sakana.new.what
```

という式を入力してください。すると、 sakana という文字列が値として得られます。

Maguro クラスは Sakana クラスのサブクラスですので、 what というメソッドを継承しています。しかし、 Maguro クラスでは、 maguro という文字列を返す同じ名前のメソッドによって what がオーバーライドされています。ですから、

```
Maguro.new.what
```

という式で、 Maguro クラスのインスタンスが持っている what を呼び出すと、 maguro という文字列が値として得られるはずです。

スーパークラスのメソッドをオーバーライドするメソッドを定義するとき、その定義の中で、オーバーライドされるスーパークラスのメソッドを使いたい、ということがしばしばあります。その場合、オーバーライドされるスーパークラスのメソッドは、 super という特殊な名前を使うことによって呼び出すことができるようになっています。

それでは、次のプログラムを入力して、 load で実行してください。

プログラムの例 tonbo.rb

```
class Konchuu
  def what
    "konchuu"
  end
end

class Tonbo < Konchuu
  def what
    super + "/tonbo"
```

```
  end
end
```

Tonbo クラスは、スーパークラスから継承した what というメソッドをオーバーライドしているわけですが、その定義の中で super を使うことによって、スーパークラスの what を呼び出しています。ですから、

```
Tonbo.new.what
```

という式で、Tonbo クラスのインスタンスが持っている what を呼び出すと、konchuu/tonbo という文字列が値として得られます。

オーバーライドされるメソッドに引数を渡したいときは、

```
super(350, "hitode")
```

というように、super の右側に丸括弧を書いて、その中に式を書きます。

引数を渡すための式を super の右側に書かなかった場合は、サブクラスのメソッドが受け取った引数をすべて渡すという意味だと解釈されます。つまり、

```
def set(a, b, c)
  super
end
```

というメソッド定義の中の super は、

```
super(a, b, c)
```

という記述と同じ意味になります。もしも、サブクラスのメソッドが引数を受け取るにもかかわらず、オーバーライドされるメソッドにまったく引数を渡したくない、という場合は、

```
super()
```

というように、空の丸括弧を書く必要があります。

3.2.6 同一のオブジェクトから構成される列

継承を利用したプログラムの例としてこれまでに紹介したものは、実用性がほとんどないものばかりでしたので、ここで、実用性が少しはあるかもしれないプログラムを紹介しておきたいと思います。

プログラムの例 sameseq.rb

```
class SameSequence
  def initialize(element, length)
    @element = element
    @length = length
  end
  def getElement
    @element
  end
  def getLength
    @length
  end
  def to_s
    if @length >= 1
      s = @element.to_s
      (@length - 1).times do
        s += "," + @element.to_s
      end
    else
      s = ""
    end
    "[" + s + "]"
  end
end

class CharSameSequence < SameSequence
  def to_s
    @element.chr * @length
  end
```

```
end

class NumericSameSequence < SameSequence
  def sum
    @element * @length
  end
  def product
    @element ** @length
  end
end
```

このプログラムの中で定義されている SameSequence というクラスは、同一のオブジェクトから構成される列をあらわすオブジェクトを生成します。

このクラスの new は、1 個目の引数として任意のオブジェクト、2 個目の引数として列の長さを受け取って、そのオブジェクトだけから構成される列をあらわすオブジェクトを戻り値として返します。たとえば、

```
a = SameSequence.new("hibari", 7)
```

という式を評価すると、hibari という文字列を 7 個並べることによってできる列をあらわすオブジェクトが a という変数に代入されます。

SameSequence クラスのインスタンスに to_s というメッセージを送ると、そのインスタンスがあらわしている列を文字列に変換したものが得られます。それでは、先ほど a という変数に代入したオブジェクトに to_s を送ってみましょう。そうすると、

```
[hibari,hibari,hibari,hibari,hibari,hibari,hibari]
```

という文字列が得られるはずです。

SameSequence のサブクラスとして定義されている CharSameSequence というクラスは、同一の文字から構成される列をあらわすオブジェクトを生成します。new に渡す 1 個目の引数は、列を構成する文字の文字コードです。たとえば、

```
b = CharSameSequence.new("="[0], 40)
```

という式を評価すると、イコール (=) という文字を 40 個並べることによってできる列をあらわすオブジェクトが b という変数に代入されます。

CharSameSequence クラスの to_s は、スーパークラスの to_s とは少し異なる動作をします。たとえば、先ほど生成したオブジェクトに to_s というメッセージを送ると、

```
========================================
```

という文字列が得られます。

SameSequence のサブクラスとして定義されている NumericSameSequence というクラスは、同一の数値から構成される列をあらわすオブジェクトを生成します。このクラスでは、レシーバーの和を求める sum というメソッドと、積を求める product というメソッドが追加されています。たとえば、

```
c = NumericSameSequence.new(2, 10)
```

という式で、2 という整数を 10 個並べることによってできる列をあらわすオブジェクトが c という変数に代入されているとすると、c.sum という式の値は 20 になって、c.product という式の値は 1024 になります。

3.3 モジュール

3.3.1 モジュールの基礎

クラスを生成する Class というクラスは、Module というクラスのサブクラスです。この Module というクラスのインスタンスは、「モジュール」(module) と呼ばれます。

モジュールは、モジュール、クラス、メソッド、メソッドの鋳型などを自分の内部に持つことができるという機能を持っています。この機能は、継承によってクラスにも受け継がれています。クラスというのは、モジュールに対して、

- オブジェクトを生成することができる。

- スーパークラスから性質を継承することができる。

という二つの機能を追加したオブジェクトだと考えることができます。

クラスが持っている鋳型ではないメソッドのことを「クラスメソッド」と呼ぶのと同じように、モジュールが持っている鋳型ではないメソッドは、「モジュールメソッド」(module method) と呼ばれます。モジュールメソッドは、それを持っているモジュールにメッセージを送ることによって呼び出すことができます。

3.3.2 モジュール定義

新しいモジュールを定義したいときや、すでに存在するモジュールに何かを追加したいときは、「モジュール定義」(module definition) と呼ばれる式を書きます。モジュール定義は、

```
module モジュール名
  式
end
```

という構文を持つ式です。

モジュール定義を評価すると、「モジュール名」のところに書かれた名前で指定されるモジュールに、「式」のところに書かれた式によって定義されたものが追加されます。存在しないモジュールの名前が指定された場合は、その名前を持つ新しいモジュールが定義されます。なお、モジュールに付ける名前は、クラスの場合と同じように、先頭の文字が英字の大文字でないといけません。

モジュールにメソッドの鋳型を追加したいときは、モジュール定義の中に普通のメソッド定義を書きます。メソッドの鋳型ではなくてモジュールメソッドを追加したいときは、

```
def モジュール名.メソッド名(仮引数名, ··· )
  式
end
```

という形のメソッド定義を書きます。

プログラムの例 umiushi.rb

```
module Umiushi
  def Umiushi.modulename
    "Umiushi"
  end
end
```

実行例

```
irb(main):001:0> load("umiushi.rb")
=> true
irb(main):002:0> Umiushi.modulename
=> "Umiushi"
```

3.3.3 モジュールのインクルード

モジュールは、自分の中にメソッドの鋳型を持つことができます。しかし、モジュールは、インスタンスを生成するという機能を持っていません。それでは、モジュールが持っているメソッドの鋳型は、いったいどうすれば利用することができるのでしょうか。

モジュールは、別のモジュールが持っているものを自分の中に取り込むという機能を持っています。モジュールが、別のモジュールが持っているものを自分の中に取り込むことを、モジュールを「インクルードする」(include) と言います。

モジュールをインクルードするという機能はクラスにも継承されていますので、クラスも、モジュールが持っているものを自分の中に取り込むことができます。モジュールをインクルードしたクラスは、インスタンスを生成するとき、モジュールから取り込んだメソッドの鋳型からもメソッドを生成します。

モジュールをインクルードしたいときは、include というメソッドを使います。モジュール定義またはクラス定義の中に、

```
include(モジュール名)
```

という形の式を書いておくと、引数で指定されたモジュールが、モジュールまたはクラスの中にインクルードされます。たとえば、

```
class Kamenote
  include(Funamushi)
end
```

というクラス定義を書いたとすると、Kamenote クラスの中に Funamushi モジュールがインクルードされます。

プログラムの例 toyshop.rb

```
module Toyshop
  def addself
    self + self
  end
end

class Numeric
  include(Toyshop)
end

class String
  include(Toyshop)
end
```

実行例

```
irb(main):001:0> load("toyshop.rb")
=> true
irb(main):002:0> 35.addself
=> 70
irb(main):003:0> "Venus".addself
=> "VenusVenus"
```

3.3.4 mixin

モジュールというのはさまざまな用途で使われるのですが、それらのうちでもっとも重要なのは、mixin という用途です。

mixin というのは、いくつかのクラスに共通して必要になる機能を、それぞれのクラスで独自に定義するのではなくて、共通する機能を別のところに作っておいて、それぞれのクラスがその機能を共有することです。mixin は、いくつかのクラスで共有したい機能をモジュールの中に作っておいて、それぞれのクラスはそのモジュールをインクルードする、という方法によって実現することができます。

機能の共有は、継承を使うことによっても実現することができますが、継承というのは、あくまで一般的なクラスの機能を特殊なクラスが受け継ぐということですから、一般的なものと特殊なものという関係が成り立っていないクラスのあいだで無理に継承を使うと、プログラムが不自然なものになってしまいます。ですから、いくつかのクラスで機能を共有したいけれども、継承を使うと、クラスとクラスとのあいだの関係が不自然になってしまう、という場合は、継承ではなくて mixin を使うほうがいいでしょう。

3.3.5 名前空間

モジュールの用途のうちで、mixin に次いで重要なのは、名前空間という用途です。

「名前空間」(namespace) というのは、名前によって識別することのできる、名前の有効範囲のことです。名前空間の中にあるものの名前は、名前空間の名前と、その中にあるものの名前とを組み合わせた名前によって指定されます。

モジュールやクラスの中にあるものは、モジュールやクラスの名前と、その中にあるものの名前とを組み合わせた名前を書かないと指定できません。ですから、モジュールやクラスは名前空間として使うことが可能です。

名前空間の中にあるものを指定したいときは、

名前空間名 : : 名前

メッセージ式	説明
Math::sqrt(x)	x の平方根を返します。x がマイナスだと例外を発生させます。
Math::exp(x)	x に対する指数関数の値 (e^x) を返します。
Math::log(x)	x の自然対数を返します。
Math::log10(x)	x の常用対数を返します。
Math::sin(x)	x のサインを返します。
Math::cos(x)	x のコサインを返します。
Math::tan(x)	x のタンジェントを返します。
Math::atan2(y, x)	y/x のアークタンジェントを返します。

表 3.1: 主要な数学関数

という形の名前を書きます。つまり、モジュールまたはクラスの名前と、その中にあるものの名前とを、コロンコロン (::) をはさんで連結することによってできる名前を書くわけです。たとえば、

```
Toolbox::Spanner
```

という名前は、Toolbox というモジュールまたはクラスの中にある Spanner という名前のものを指定します。

ちなみに、モジュールメソッドやクラスメソッドを呼び出す式を、ドットの代わりにコロンコロンを使って書くことも可能です。

プログラムの例 ocean.rb

```
module Ocean
  class Namako
    def Namako.classname
      "Namako"
    end
  end
end
```

実行例

```
irb(main):001:0> load("ocean.rb")
=> true
irb(main):002:0> Ocean::Namako::classname
=> "Namako"
```

プログラムの中では、さまざまなものが、それに与えられた名前によって識別されます。ものに名前を与える場合には、「異なるものには異なる名前を与えないといけない」という原則を守る必要があります。しかし、プログラムの規模が大きくなればなるほど、その原則を守ることが難しくなります。気付かないうちに、異なるものに同じ名前が与えられているという状態、すなわち「名前の衝突」が発生することがあります。

ですから、大規模なプログラムを書く場合には、名前空間を活用することが必要です。そうすることによって、名前の衝突が発生することを未然に防ぐことができます。

3.3.6 組み込みモジュール

Ruby の処理系の中にはさまざまなクラスが組み込まれているわけですが、クラスだけではなくて、さまざまなモジュールも組み込まれています。処理系の中に組み込まれているモジュールは、「組み込みモジュール」(built-in module) と呼ばれます。

組み込みモジュールは、mixin に利用されるものと、名前空間として使われているものに分類することができます。

mixin に利用される組み込みモジュールの例としては、Comparable というモジュールがあります。これは、<、>、<=、>= などの、大小関係を調べる述語を持っているモジュールです。数値のクラスや文字列のクラスは、これらの述語を独自に定義しているのではなくて、Comparable をインクルードすることによって自分の中に取り込んでいるのです。

モジュールを名前空間として使っている組み込みモジュールの例としては、Math というモジュールがあります。これは、表 3.1 に示されているような、さまざまな数学関数を持っているモジュールです。たとえば、

```
Math::sqrt(2)
```

という式を評価すると、$\sqrt{2}$ が得られます。

3.4 ライブラリー

3.4.1 ライブラリーの基礎

プログラムというものは、それ自体の動作を利用することを目的として書かれることもあれば、別のプログラムに機能を提供することを目的として書かれることもあります。後者のプログラムは、「ライブラリー」(library) と呼ばれます。

ライブラリーは、自分で書いて自分で使うこともできますし、誰かが書いたものを入手して使うこともできます。

Ruby の場合、ライブラリーによって提供される機能を利用するためには、ライブラリーを読み込むという処理をする必要があります。

ライブラリーは、require という関数的メソッドを呼び出すことによって読み込むことができます。このメソッドは、引数としてパス名を受け取って、そのパス名で指定されたファイルに格納されているライブラリーを読み込みます。ただし、指定されたライブラリーがすでに読み込まれている場合、それを重複して読み込むということはしません。

相対パス名でライブラリーを指定した場合、require は、あらかじめライブラリーの置き場所として設定されたディレクトリを探索します。

ライブラリーのファイル名の拡張子が .rb の場合、その拡張子は省略することができます。ですから、namako.rb というファイル名のライブラリーは、

```
require("namako")
```

という式を書くことによって読み込むことができます。

それでは、ライブラリーとして利用されるプログラムと、それを利用するプログラムとを実際に書いてみましょう。

まず、ライブラリーとして利用されるプログラムを準備します。次のプログラムは、レシーバーが素数ならば真を返して、そうでなければ偽を返す、prime という述語を整数に追加します。

プログラムの例 prime.rb

```
class Integer
  def prime
    if self <= 1
      false
    else
      i = 2
      found = false
      while ! found && i*i <= self
        if self%i == 0
          found = true
        else
          i += 1
        end
      end
      ! found
    end
  end
end
```

そして次は、上のプログラムをライブラリーとして利用するプログラムです。次のプログラムは、レシーバーを番号とする素数を求める thprime というメソッドを整数に追加します（素数の番号というのは、素数を小さなものから大きなものへと並べたときの順番をあらわすプラスの整数のことです）。

プログラムの例 thprime.rb

```
require("prime")
```

ライブラリー名	説明
rational	有理数をあらわす Rational クラスを定義します。
complex	複素数をあらわす Complex クラスを定義します。
matrix	行列をあらわす Matrix クラスを定義します。
date	日付をあらわす Date クラスを定義します。
kconv	文字コードを変換する各種のメソッドを定義します。
cgi	CGI をあらわす CGI クラスを定義します。
socket	ソケットをあらわす各種のクラスを定義します。
tk	GUI を作るための各種のクラスを定義します。

表 3.2: Ruby の標準ライブラリーのうちの主要なもの

```
class Integer
  def thprime
    i = 1
    number = 0
    while number < self
      i += 1
      if i.prime
        number += 1
      end
    end
    i
  end
end
```

実行例

```
irb(main):001:0> load("thprime.rb")
=> true
irb(main):002:0> 555.thprime
=> 4019
```

3.4.2　標準ライブラリー

処理系に組み込んだほうがいいと思われるほど基本的な機能ではないけれども、きわめて汎用性が高いと思われる機能は、普通、処理系とともに配布されるライブラリーによって提供されます。処理系とともに配布されるライブラリーの集まりは、「標準ライブラリー」(standard library) と呼ばれます。

Ruby の処理系も、さまざまなライブラリーとともに配布されています。表 3.2 に、Ruby の標準ライブラリーに含まれているライブラリーのうちの主要なものを示しておきます。

第 4 章　組み込みクラス

4.1　例外

4.1.1　例外の基礎

Ruby の処理系には、Object、Class、Fixnum、Float、String などの、「組み込みクラス」と呼ばれるさまざまなクラスが最初から組み込まれています。この章では、Ruby の組み込みクラスのうちで、まだ登場していないものについて説明していきたいと思います。

まず最初に、「例外」と呼ばれるオブジェクトを生成するクラスを紹介しましょう。

第 2.11 節で、レシーバーの階乗を求めるメソッドを、

```
class Integer
  def factorial
    if self >= 1
```

```
        self * (self - 1).factorial
      else
        1
      end
    end
  end
```

と定義しましたが、これは、厳密には正しい定義とは言えません。なぜなら、マイナスの整数に対する階乗というものは定義されていないにもかかわらず、このメソッドは、レシーバーがマイナスの整数であっても 1 という戻り値を返すからです。このメソッドの定義をもっと正しいものにするためには、レシーバーがマイナスの整数だという不都合な事態に遭遇した場合はそのことを報告するように改良しないといけません。さて、それでは、何らかの不都合な事態に遭遇した場合にそのことを報告する、という動作は、いったいどう書けばいいのでしょうか。

何らかの不都合な事態に遭遇した場合に、それを報告するために採用されるもっとも一般的な手段は、「例外」(exception) と呼ばれるオブジェクトを生成する、というものです。例外というのは、実行中のメソッドの動作を終了させて、メソッドを呼び出したものに対して、遭遇した不都合な事態を報告する、という機能を持つオブジェクトのことです。例外を生成することを、例外を「発生させる」(raise) と言います。

例外は、実行中のメソッドの動作を終了させるという機能を持っています。ですから、例外をそのまま放置すると、プログラム全体が終了してしまいます。例外が発生したとき、プログラム全体を終了させるのではなくて、不都合な事態に対する何らかの処置を実行したいときは、例外がメソッドの動作を終了させようとするのを抑止する必要があります。例外によるメソッドの動作の終了を抑止することを、例外を「捕獲する」(catch) と言います。

4.1.2 例外クラス

例外は、`Exception` という組み込みクラス、またはそれを継承するクラスから生成されるオブジェクトです。そのような、例外を生成するクラスは、「例外クラス」(exception class) と呼ばれます。Ruby の処理系には、`Exception` だけではなくて、それを継承するさまざまな例外クラスが組み込まれています。

たとえば、Ruby の処理系には、`ZeroDivisionError` という例外クラスが組み込まれています。整数のオブジェクトが持っている `/` という除算のメソッドは、除数がゼロだった場合、この例外クラスの例外を発生させます[1]。

それでは、`5/0` のような、整数をゼロで除算する式を irb に入力してみましょう。そうすると、次のようになります。

```
irb(main):001:0> 5/0
ZeroDivisionError: divided by 0
        from (irb):1:in `/'
        from (irb):1
irb(main):002:0>
```

このように、irb は、例外が発生すると、たいていの場合、メッセージを出力したのち、次のプロンプトを出力します。例外が発生したときに irb が出力するメッセージは、発生した例外のクラス名、例外が持っている文字列、そして例外が発生した場所を示しています。

irb は、すべての例外を捕獲するわけではありません。irb が捕獲するのは、`StandardError` という例外クラス、またはそれを継承するクラスの例外だけです。整数をゼロで除算したときに発生する例外が捕獲されるのは、それが `StandardError` を継承しているクラスの例外だからです。

irb に入力された式が評価されている途中で、irb によって捕獲されない例外が発生した場合、その例外は、irb を終了させることになります。実は、`exit` という式を irb に入力することによって irb を終了させることができるのは、`exit` という関数的メソッドが、irb によって捕獲されない `SystemExit` というクラスの例外を発生させるからです。

4.1.3 例外の発生

例外を発生させたいときは、`raise` という関数的メソッドを使います。

引数を何も渡さないで `raise` を呼び出すと、`raise` は、`RuntimeError` という例外クラスの例外を発生させます（`RuntimeError` は `StandardError` のサブクラスです）。

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

[1] 浮動小数点数の `/` は、ゼロで除算をしても例外は発生させません。

```
    RuntimeError:
            from (irb):1
    irb(main):002:0>
```

例外には、遭遇した不都合な事態について説明するために、1 個の文字列を持たせることができます。例外に文字列を持たせたいときは、持たせたい文字列を引数として raise に渡します。

```
    irb(main):001:0> raise("somothing wrong")
    RuntimeError: somothing wrong
            from (irb):1
    irb(main):002:0>
```

引数として例外クラスを渡して raise を呼び出すと、raise は、受け取ったクラスの例外を発生させます。

```
    irb(main):001:0> raise(ZeroDivisionError)
    ZeroDivisionError: ZeroDivisionError
            from (irb):1
    irb(main):002:0>
```

余談ですが、irb を終了させることのできるメソッドは exit だけではありません。raise を使って irb を終了させることも可能です。

```
    irb(main):001:0> raise(SystemExit)
    $
```

例外クラスを指定して例外を発生させて、さらに、その例外に文字列を持たせたい、という場合は、raise に対して、1 個目の引数として例外クラス、2 個目の引数として文字列を渡します。

```
    irb(main):001:0> raise(StandardError, "How terrible it is!")
    StandardError: How terrible it is!
            from (irb):1
    irb(main):002:0>
```

次のプログラムは、レシーバーの階乗を求めるメソッドの定義を、レシーバーがマイナスだった場合は StandardError という例外クラスの例外を発生させるように改良したものです。

プログラムの例　fact2.rb

```
class Integer
  def factorial
    if self >= 1
      self * (self - 1).factorial
    elsif self == 0
      1
    else
      raise(StandardError,
        "factorial is not defined to minus integer")
    end
  end
end
```

実行例

```
irb(main):001:0> load("fact2.rb")
=> true
irb(main):002:0> 30.factorial
=> 265252859812191058636308480000000
irb(main):003:0> -30.factorial
StandardError: factorial is not defined to minus integer
        from ./fact2.rb:8:in `factorial'
        from (irb):3
irb(main):004:0>
```

4.1.4　例外の捕獲

例外を捕獲するメソッドを定義したいときは、

```
def メソッド名(仮引数名, ··· )
  式
rescue 例外クラス名, ···
  式
end
```

という形のメソッド定義を書きます。この形のメソッド定義の中に含まれている、

```
rescue 例外クラス名, ···
  式
```

という形の部分は、「rescue 節」(rescue clause) と呼ばれます。

rescue 節の上に書かれた式が評価されているときに、rescue の右側で指定されたクラス、またはそれを継承するクラスの例外が発生したとすると、その例外は捕獲されて、rescue 節の中の式が評価されます。指定された例外が発生しなかった場合、rescue 節の中の式は評価されません。

プログラムの例 divide.rb

```
class Integer
  def divide(n)
    (self / n).display
    "\n".display
  rescue ZeroDivisionError
    "ゼロでは除算できません。\n".display
  end
end
```

実行例

```
irb(main):001:0> load("divide.rb")
=> true
irb(main):002:0> 56.divide(7)
8
=> nil
irb(main):003:0> 56.divide(0)
ゼロでは除算できません。
=> nil
irb(main):004:0>
```

rescue 節は、メソッド定義の中に何個でも好きなだけ書くことができます。そうすることによって、発生した例外のクラスごとに異なる動作を実行することが可能になります。

プログラムの例 divide2.rb

```
class Integer
  def divide(n)
    (self / n).display
    "\n".display
  rescue ZeroDivisionError
    "ゼロでは除算できません。\n".display
  rescue TypeError
    "引数のクラスが間違っています。\n".display
  end
end
```

実行例

```
irb(main):001:0> load("divide2.rb")
=> true
irb(main):002:0> 56.divide(0)
ゼロでは除算できません。
=> nil
irb(main):003:0> 56.divide("seven")
```

```
引数のクラスが間違っています。
=> nil
irb(main):004:0>
```

例外クラスは、Exception を根とする木の構造を持っています。根に近い位置にあるクラスの名前を rescue の右側に書くことによって、そのクラスから枝分かれしているすべてのクラスの例外を一網打尽にすることができます。

プログラムの例　divide3.rb

```
class Integer
  def divide(n)
    (self / n).display
    "\n".display
  rescue StandardError
    "例外が発生しました。\n".display
  end
end
```

実行例

```
irb(main):001:0> load("divide3.rb")
=> true
irb(main):002:0> 56.divide(0)
例外が発生しました。
=> nil
irb(main):003:0> 56.divide("seven")
例外が発生しました。
=> nil
irb(main):004:0>
```

ちなみに、rescue の右側の例外クラス名を省略すると、StandardError という例外クラスを指定したと解釈されます。ですから、上のプログラムの rescue 節は、例外クラス名を省略したとしても同じ意味になります。

メソッド定義の全体ではなくて、その一部分で発生した例外だけを捕獲したいというときは、その部分に「begin 式」(begin expression) と呼ばれる式を書きます。

begin 式は、

```
begin
    式
rescue 例外クラス名, ···
    式
end
```

と書きます。この形の式を評価すると、まず、 rescue 節の上に書かれた式が評価されます。その途中で、 rescue 節で指定されたクラスの例外が発生した場合は、 rescue 節の中の式が評価されます。

4.1.5　後始末

メソッドを定義するとき、例外が発生したか発生しなかったかにかかわらず、かならず後始末を実行してから動作が終了するようにしたい、と思うことがしばしばあります。そのような後始末の動作は、「ensure 節」(ensure clause) と呼ばれるものをメソッド定義または begin 式の中に書くことによって記述することができます。

ensure 節は、

```
ensure
    式
```

と書きます。 ensure 節の中の式は、例外が発生したとしても発生しなかったとしても、かならず評価されます。

プログラムの例　divide4.rb

```
class Integer
  def divide(n)
    (self / n).display
    "\n".display
  ensure
    "除算はこれで終了です。\n".display
  end
end
```

実行例

```
irb(main):001:0> load("divide4.rb")
=> true
irb(main):002:0> 56.divide(7)
8
除算はこれで終了です。
=> nil
irb(main):003:0> 56.divide(0)
除算はこれで終了です。
ZeroDivisionError: divided by 0
        from ./divide4.rb:3:in `/'
        from ./divide4.rb:3:in `divide'
        from (irb):3
irb(main):004:0>
```

例外が発生した場合はそれを捕獲して、なおかつ、例外が発生したか発生しなかったかにかかわらず後始末も実行したい、というときは、 rescue 節と ensure 節の両方をメソッド定義または begin 式の中に書きます。ただし、 rescue 節と ensure 節は、 rescue 節が上で ensure 節が下という順番で書かないといけません。

プログラムの例 divide5.rb

```
class Integer
  def divide(n)
    (self / n).display
    "\n".display
  rescue ZeroDivisionError
    "ゼロでは除算できません。\n".display
  ensure
    "除算はこれで終了です。\n".display
  end
end
```

実行例

```
irb(main):001:0> load("divide5.rb")
=> true
irb(main):002:0> 56.divide(7)
8
除算はこれで終了です。
=> nil
irb(main):003:0> 56.divide(0)
ゼロでは除算できません。
除算はこれで終了です。
=> nil
irb(main):004:0>
```

4.1.6 例外が持っている文字列

例外を発生させる方法について説明したときに、例外には文字列を持たせることができるという話をしましたが、それでは、例外が持っている文字列を利用するためにはどうすればいいのでしょうか。

そのためには、まず、発生した例外を変数に代入する必要があります。例外を変数に代入したいときは、 rescue の右側の例外クラス名のさらに右側に、

=> 変数名

という形のものを書きます。そうすると、その中で指定された変数に例外が代入されます。たとえば、

```
rescue StandardError => e
  (e.class.to_s + "\n").display
end
```

という rescue 節を書くことによって、発生した例外のクラス名を出力することができます。

例外が持っている文字列を取り出したいときは、message というメソッドを使います。このメソッドは、レシーバーが持っている文字列を戻り値として返します。

プログラムの例　divide6.rb

```
class Integer
  def divide(n)
    (self / n).display
    "\n".display
  rescue StandardError => e
    ("例外クラス: " + e.class.to_s + "\n" +
    "例外が持っている文字列: " + e.message + "\n").display
  end
end
```

実行例

```
irb(main):001:0> load("divide6.rb")
=> true
irb(main):002:0> 56.divide(0)
例外クラス: ZeroDivisionError
例外が持っている文字列: divided by 0
=> nil
irb(main):003:0> 56.divide("seven")
例外クラス: TypeError
例外が持っている文字列: String can't be coerced into Fixnum
=> nil
irb(main):004:0>
```

4.1.7　独自の例外クラス

Ruby のインタプリタの中には、さまざまな例外クラスが組み込まれているわけですが、例外クラスはそれらだけではありません。独自の例外クラスを定義するということも可能です。

独自の例外クラスを定義したいときは、ただ単に、すでに存在する例外クラスのどれかをスーパークラスとして指定したクラス定義を書けばいいだけです。独自の例外クラスにメソッドの鋳型を追加しなくていいならば、クラス定義の中にメソッド定義を書く必要はありません。たとえば、

```
class OriginalError < StandardError
end
```

というクラス定義を書くことによって、OriginalError という独自の例外クラスを定義することができます。

次のプログラムは、OutOfDomain という独自の例外クラスを定義して、レシーバーの階乗を求めるメソッドを、レシーバーがマイナスだった場合はその例外クラスの例外を発生させるように定義しています。

プログラムの例　fact3.rb

```
class OutOfDomain < StandardError
end

class Integer
  def factorial
    if self >= 1
      self * (self - 1).factorial
    elsif self == 0
      1
    else
      raise(OutOfDomain,
```

```
        "factorial is not defined to minus integer")
    end
  end
end
```

実行例

```
irb(main):001:0> load("fact3.rb")
true
irb(main):002:0> 30.factorial
265252859812191058636308480000000
irb(main):003:0> -30.factorial
OutOfDomain: factorial is not defined to minus integer
        from ./fact3.rb:11:in `factorial'
        from (irb):3
irb(main):004:0>
```

4.2 配列

4.2.1 配列の基礎

Ruby のプログラムは、「配列」(array) と呼ばれるオブジェクトを扱うことができます。配列は、 Array というクラスのインスタンスです。

配列というのは、オブジェクトを何個でも好きなだけ並べてできる列を内部に格納することができるオブジェクトのことです（厳密に言うと、配列の中に格納されるのはオブジェクトそのものではなくて、オブジェクトを指し示す「参照」(reference) と呼ばれるデータです)。

配列に格納されているそれぞれのオブジェクトのことを、その配列の「要素」(element) と呼びます。そして、配列に格納されている要素の個数のことを、その配列の「大きさ」(size) と呼びます。

配列の大きさは、0 の場合もあります。そのような、要素をまったく持っていない配列は、「空配列」(empty array) と呼ばれます。

配列のような、任意の個数のオブジェクトを自分の中に格納することのできるオブジェクトは、「コンテナ」(container) と呼ばれます。

4.2.2 列挙による配列の生成

配列は、さまざまな方法で生成することができます。配列を生成する方法のひとつは、それを構成する要素を列挙する、というものです。

要素を列挙することによって配列を生成したいときは、

[式, 式, 式, ···]

という形の式を書きます。つまり、式をコンマ (,) で区切って並べて、その全体を角括弧 ([]) で囲むわけです。この形の式を評価すると、それを構成するそれぞれの式が評価されて、それらの式の値を並べることによってできる配列が、値として得られます。要素の順番は、式が並んでいる順番と同じです。たとえば、

```
[38, -21, 0.47, "hitode"]
```

という式を評価すると、 38 と -21 と 0.47 と "hitode" というそれぞれのリテラルがあらわしているオブジェクトを、この順番で並べることによってできる配列が、値として得られます。

角括弧だけの式、つまり [] という式を評価すると、値として空配列が得られます。

4.2.3 new による配列の生成

配列は、 Array クラスが持っている new というメソッドを呼び出すことによって生成することもできます。

Array クラスの new は、引数を何も渡さないで呼び出した場合、空配列を生成して、それを戻り値として返します。ですから、

```
Array.new
```

という式を評価すると、その値として空配列が得られます。

メッセージ式	説明
a.size	a の大きさを返します。
a.shift	a から先頭の要素を削除して、その要素を返します。
a.reverse	a の要素を逆の順序に並べ替えてできる配列を返します。
a.sort	a の要素を大きさの順序で並べ替えてできる配列を返します。
a.uniq	a の重複する要素をひとつだけにした配列を返します。
a.join(s)	a の要素を文字列 s で区切って連結してできる文字列を返します。
a.empty?	a が空配列ならば true、そうでなければ false を返します。
a.member?(o)	o が a の要素ならば true、そうでなければ false を返します。

表 4.1: 配列の基本的なメソッド

Array の new に、引数として 0 またはプラスの整数を渡すと、引数で指定された大きさを持つ、すべての要素が nil であるような配列が生成されます。

```
irb(main):001:0> Array.new(10)
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]
```

Array の new に、1 個目の引数として 0 またはプラスの整数を渡して、2 個目の引数として何らかのオブジェクトを渡すと、1 個目の引数で指定された大きさを持つ、すべての要素が 2 個目の引数であるような配列が生成されます。

```
irb(main):001:0> Array.new(10, 0)
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
irb(main):002:0> Array.new(3, "namekuji")
=> ["namekuji", "namekuji", "namekuji"]
```

4.2.4　範囲からの配列の生成

範囲によってあらわされているオブジェクトの列から配列を生成する、ということも可能です。

範囲から配列を生成したいときは、範囲が持っている to_a というメソッドを使います。これは、レシーバーに含まれているそれぞれのオブジェクトを要素とする配列を生成して、その結果を戻り値として返すメソッドです。

```
irb(main):001:0> (20..30).to_a
=> [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
irb(main):002:0> ("ax".."bc").to_a
=> ["ax", "ay", "az", "ba", "bb", "bc"]
```

4.2.5　文字列からの配列の生成

文字列をいくつかの部分に分解して、それぞれの部分から構成される配列を生成する、ということも可能です。

文字列から配列を生成したいときは、文字列が持っている split というメソッドを使います。これは、連続する空白でレシーバーを区切って、それぞれの部分から構成される配列を戻り値として返すメソッドです。

```
irb(main):001:0> "  above us  only   sky  ".split
=> ["above", "us", "only", "sky"]
```

引数として文字列を渡すことによって、何で区切るかということを指定することもできます。

```
irb(main):001:0> "www.paradise.ac.jp".split(".")
=> ["www", "paradise", "ac", "jp"]
```

引数として空文字列を渡すことによって、長さが 1 の文字列に分解することもできます。

```
irb(main):001:0> "animism".split("")
=> ["a", "n", "i", "m", "i", "s", "m"]
```

演算子式	説明
a << o	a の末尾に o を追加して、その結果を返します。
a + b	a の右側に b を連結した配列を返します。
a - b	b の要素を a から取り除いた配列を返します。
a * n	a を n 回繰り返して連結した配列を返します。
a & b	a と b に共通する要素から構成される配列を返します。
a | b	a と b のどちらかに含まれている要素から構成される配列を返します。
a[i]	a の i 番目の要素を返します。
a[i..j]	a の i 番目から j 番目までの部分配列を返します。
a[i, n]	a の i 番目から始まる長さが n の部分配列を返します。
a[i] = o	a の i 番目の要素を o に置き換えます。
a[i..j] = b	a の i 番目から j 番目までの部分配列を b に置き換えます。
a[i, n] = b	a の i 番目から始まる長さが n の部分配列を b に置き換えます。

表 4.2: 配列の演算

4.2.6 配列の基本的なメソッド

配列というオブジェクトは、自分自身を取り扱うためのさまざまなメソッドを持っています。表 4.1 は、配列が持っているメソッドのうちの基本的なものを示しています。

irb を使って、配列が持っているメソッドの動作を確かめてみることにしましょう。たとえば、

```
[58, 33, 27, 63].join("/")
```

という式を irb に入力してみてください。そうすると、join は、レシーバーを構成するそれぞれの要素を文字列に変換して、それらをスラッシュ(/)で区切って連結することによってできる文字列を戻り値として返します。ですから、

```
"58/33/27/63"
```

という文字列が、式の値として出力されます。

shift というメソッドは、レシーバーの先頭の要素を削除して、その要素を戻り値として返します。レシーバー自体を変化させるという点に注意してください。

shift の動作を、実際に確かめてみましょう。まず、

```
a = [74, 21, 60, 83]
```

という式を irb に入力することによって、a という変数に配列を代入してください。次に、

```
a.shift
```

という式で、shift を呼び出してください。すると、shift は、a が指し示している配列から先頭にある 74 という要素を削除して、その要素を戻り値として返します。次に、a という変数名を入力してください。すると、先頭の要素を削除したのちの配列、つまり、

```
[21, 60, 83]
```

という配列が出力されるはずです。

4.2.7 配列への要素の追加

配列は、表 4.2 に示されているようなさまざまな演算を持っています。

<< は、レシーバーの末尾に要素を追加して、そののちのレシーバーを戻り値として返す、という動作をする演算です。<< も、shift と同じように、レシーバーを変化させるメソッドです。

irb を使って、<< の動作を実際に確かめてみましょう。まず、

```
a = Array.new(4, "kani")
```

という式を入力してください。すると、

```
["kani", "kani", "kani", "kani"]
```

という配列が a という変数に代入されます。次に、

```
a << "fujitsubo"
```

という式を入力してください。そうすると、 a が指し示している配列の末尾に "fujitsubo" という文字列が追加されて、そののちの配列が値として得られます。ですから、

```
["kani", "kani", "kani", "kani", "fujitsubo"]
```

という配列が式の値として出力されます。さらに、 a が指し示している配列が変化している、ということも確認しておきましょう。 a という変数名を irb に入力することによって、それが指し示している配列を irb に出力させてみてください。

それでは、 << を使ってプログラムを書いてみることにしましょう。次のプログラムの中で定義されている sequence は、等差数列になっている配列を生成するメソッドです。

プログラムの例　sequen.rb

```
class Numeric
  def sequence(e, s)
    a = []
    step(e, s) { |x| a << x }
    a
  end
end
```

sequence は、b と e と s が数値だとするとき、

```
b.sequence(e, s)
```

というメッセージ式で呼び出すと、b を初項、s を公差とする、e を超えない範囲（s がマイナスの場合は、e を下回らない範囲）の等差数列になっている配列を、戻り値として返します。

実行例

```
irb(main):001:0> load("sequen.rb")
=> true
irb(main):002:0> 0.sequence(100, 15)
=> [0, 15, 30, 45, 60, 75, 90]
irb(main):003:0> 100.sequence(0, -15)
=> [100, 85, 70, 55, 40, 25, 10]
irb(main):004:0> 0.0.sequence(10.0, 1.5)
=> [0.0, 1.5, 3.0, 4.5, 6.0, 7.5, 9.0]
```

4.2.8　配列からの要素の取り出し

次に、配列から要素を取り出す方法について説明しましょう。

配列を構成するそれぞれの要素は、先頭から何番目にあるかという番号を持っています。ただし、要素の番号は、先頭を 0 番目と数えます。たとえば、

```
["spring", "summer", "autumn", "winter"]
```

という配列の場合、 "spring" は 0 番目で、 "winter" は 3 番目です。

番号を指定することによって配列から要素を取り出したいときは、配列が持っている [] という演算を使います。a が配列で、i が番号だとするとき、

```
a[i]
```

という式を評価すると、a の i 番目の要素が値として得られます。たとえば、

```
["spring", "summer", "autumn", "winter"][2]
```

という式を評価すると、値として "autumn" が得られます。

配列を構成するそれぞれの要素は、0 またはプラスの番号のほかに、マイナスの番号も持っています。マイナスの番号は、配列の末尾から先頭へ向かって、-1、-2、-3、-4、・・・、と付けられています。 [] は、マイナスの番号で要素を指定することもできます。たとえば、

```
["spring", "summer", "autumn", "winter"][-3]
```

という式を評価すると、値として "summer" が得られます。

[] は、レシーバーの中に存在していない要素の番号が指定された場合は、戻り値として nil を返します。たとえば、

```
["spring", "summer", "autumn", "winter"][4]
```

という式の値は、nil になります。

4.2.9 配列の要素の置き換え

配列が持っている []= という演算を使うことによって、番号で指定された配列の要素を別のオブジェクトに置き換える、ということができます。a が配列で、i が番号で、o がオブジェクトだとするとき、

```
a[i] = o
```

という式を評価すると、a の i 番目の要素が o に置き換わります。[]= も、shift や << と同じように、レシーバーを変化させるメソッドです。

irb を使って、[]= の動作を実際に試してみましょう。まず、

```
a = Array.new(6, "namako")
```

という式を入力してください。すると、

```
["namako", "namako", "namako", "namako", "namako", "namako"]
```

という配列が a という変数に代入されます。次に、

```
a[3] = "umiushi"
```

という式を入力してください。そうすると、a が指し示している配列の 3 番目の要素が "umiushi" に置き換わります。それでは、そうなったかどうかを確かめてみましょう。a という変数名を入力してください。すると、

```
["namako", "namako", "namako", "umiushi", "namako", "namako"]
```

という配列が出力されるはずです。

4.2.10 配列と代入演算

第 2.4 節で説明したように、変数というのはオブジェクトに付ける名札のようなものです。オブジェクトに変数という名札を付けることを、変数にオブジェクトを「代入する」と言います。変数にオブジェクトを代入したいときは、= という演算を使います。

= に可能な動作は、ひとつの変数にひとつのオブジェクトを代入する、ということだけではありません。実は、いくつかの変数のそれぞれに異なるオブジェクトを代入する、ということも可能なのです。そのような代入は、「多重代入」(multiple assignment) と呼ばれます。

多重代入を実行したいときは、

```
変数名, ··· = 式, ···
```

というように、変数名をコンマで区切って並べたものを = の左辺に書いて、式をコンマで区切って並べたものを右辺に書きます。このような式を評価すると、それぞれの式の値が、並んでいる順番のとおりに、対応する変数に代入されます。そして、式全体の値は、それぞれの式の値から構成される配列になります。

実際に試してみましょう。まず、

```
a, b, c = 63, 42, 21
```

という式を irb に入力してみてください。すると、その式の値として、

```
[63, 42, 21]
```

という配列が得られるわけですが、それだけではなくて、a に 63、b に 42、c に 21 が代入されます。それでは、それらの変数名を入力することによって、そうなっているかどうかを確かめてみてください。

配列を構成するそれぞれの要素を変数に多重代入する、ということも可能です。変数名をコンマで区切って並べたものを = の左辺に書いて、配列を求める式を右辺に書くと、その配列のそれぞれの要素が変数に多重代入されます。たとえば、

```
d, e, f = [93, 62, 31]
```

という式を評価すると、d に 93、e に 62、f に 31 が代入されます。

なお、右辺の要素の個数が左辺の変数名の個数よりも少ない場合、対応する要素のない変数には nil が代入されます。

4.2.11　配列のイテレーター

次に、配列が持っているイテレーターを紹介することにしましょう。

配列が持っているイテレーターのうちでもっとも動作が単純なのは、each というイテレーターです。each は、レシーバーを構成するそれぞれの要素を順番にブロックに渡してブロックを実行します。たとえば、

```
[78, "yadokari", 0.462].each do |e|
  e.display
  "\n".display
end
```

という式を評価すると、配列を構成するそれぞれの要素が出力されます。ちなみに、each は、レシーバーをそのまま戻り値として返します。

次のプログラムの中で定義されている sum というメソッドは、レシーバーが数値の配列ならば、それらの数値を合計した結果を戻り値として返します。

プログラムの例　sum.rb

```
class Array
  def sum
    s = 0
    each { |n| s += n }
    s
  end
end
```

実行例

```
irb(main):001:0> load("sum.rb")
=> true
irb(main):002:0> [71, 22, 63, 98].sum
=> 254
```

配列が持っている each 以外のイテレーターも、レシーバーを構成するそれぞれの要素を順番にブロックに渡してブロックを実行する、という動作の基本は each と共通です。

map というイテレーターは、ブロックの中の式の値から構成される配列を戻り値として返します。たとえば、

```
[5, 4, 7, 2, 8].map { |n| n * n }
```

という式を評価すると、

```
[25, 16, 49, 4, 64]
```

というように、レシーバーを構成するそれぞれの数値を 2 乗したものから構成される配列が値として得られます。

次のプログラムの中で定義されている mapadd というメソッドは、レシーバーの要素と引数が数値ならば、それぞれの要素と引数とを加算した結果から構成される配列を返します。また、レシーバーの要素と引数が文字列ならば、それぞれの要素と引数とを連結した結果から構成される配列を返します。

プログラムの例　mapadd.rb

```
class Array
  def mapadd(a)
    map { |e| e + a }
  end
end
```

実行例

```
irb(main):001:0> load("mapadd.rb")
=> true
irb(main):002:0> [47, 33, 28, 16].mapadd(1000)
```

```
=> [1047, 1033, 1028, 1016]
irb(main):003:0> ["ebi", "kani", "tako"].mapadd("modoki")
=> ["ebimodoki", "kanimodoki", "takomodoki"]
```

4.2.12 コマンドライン引数

シェルは、読み込んだコマンドを空白で区切っていくつかの単語に分解して、先頭の単語をプログラムの名前とみなして、その名前のプログラムを起動します。そして、残りの単語を、起動したプログラムに渡します。シェルからプログラムへ渡されるそれぞれの単語は、「コマンドライン引数」(command line argument) と呼ばれます。

ruby という、対話型ではない Ruby のインタプリタは、任意の個数のコマンドライン引数をシェルから受け取ることができます。ruby は、受け取ったコマンドライン引数のうちの先頭の単語をパス名とみなして、そのパス名で指定されたファイルの中のプログラムを実行します。そして、残りのコマンドライン引数を、自分が起動したプログラムに渡します。たとえば、

```
ruby program.rb ringo mikan ichigo
```

というコマンドで ruby を起動したとすると、ruby は、`program.rb` というファイルに格納されているプログラムを起動して、`ringo`、`mikan`、`ichigo` という 3 個のコマンドライン引数を、そのプログラムに渡します。

ruby は、それぞれのコマンドライン引数を要素とする配列を作って、その配列に `ARGV` という名前を付けて、その配列をプログラムに渡します。ですから、Ruby のプログラムの中に `ARGV` と書くと、それは、コマンドライン引数から構成される配列を意味することになります。

それでは、受け取ったコマンドライン引数をすべて出力するプログラムを書いてみましょう。

プログラムの例 `echo.rb`

```
ARGV.join(" ").display
"\n".display
```

実行例

```
$ ruby echo.rb kaki ichijiku biwa momo kuri
kaki ichijiku biwa momo kuri
```

`ARGV` を構成する要素はすべて文字列ですので、それを数値として扱いたい場合は、`to_i` や `to_f` などを使って、それを数値に変換する必要があります。

次のプログラムは、コマンドライン引数として 1 個の整数を受け取って、1 からその整数までのそれぞれの整数を出力します。

プログラムの例 `oneton.rb`

```
if ARGV.size == 1
  ARGV[0].to_i.times do |i|
    (i + 1).display
    " ".display
  end
  "\n".display
else
  "使い方: ruby oneton.rb 整数\n".display
end
```

実行例

```
$ ruby oneton.rb
使い方: ruby oneton.rb 整数
$ ruby oneton.rb 23
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

なお、決まった個数のコマンドライン引数を受け取るプログラムを書くときは、このプログラムのように、コマンドライン引数の個数を確認して、それが正しくない場合はプログラムの使い方 (usage) を出力するようにしておくと、使う人に対して親切なプログラムになります。

4.3　ハッシュ

4.3.1　ハッシュの基礎

Ruby の組み込みクラスの中には、コンテナを生成するクラスが、`Array` のほかにもうひとつあります。それは、`Hash` というクラスです。このクラスは、「ハッシュ」(hash) と呼ばれるオブジェクトを生成します[2]。

ハッシュというのはコンテナの一種ですから、任意の個数のオブジェクトをその中に格納することができます。それらのオブジェクトは、ハッシュの「要素」(element) と呼ばれます。ただし、ハッシュのひとつの要素は、ひとつのオブジェクトではありません。ハッシュの場合は、二つのオブジェクトがペアになったものがひとつの要素になるのです。ハッシュの要素を構成する二つのオブジェクトのそれぞれは、「キー」(key) と「値」(value) と呼ばれます。

配列は、その中で要素がどんな順序で並んでいるかという情報を持っています。ですから、配列のそれぞれの要素は、それが何番目にあるかという番号によって指定することができます。それに対して、ハッシュは、要素がどんな順序で並んでいるかという情報を持っていません。ですから、ハッシュのそれぞれの要素を、それが何番目にあるかという番号で指定することはできません。それでは、ハッシュの要素を指定したいときは、いったいどうすればいいのでしょうか。

ハッシュには、それぞれの要素は互いに異なるキーを持っていないといけない、という制約が課されています。この制約のお蔭で、ハッシュの要素は、その要素のキーによって指定することができます。

ちなみに、ハッシュの要素の値に関しては、互いに異なっていないといけないという制約はありません。ですから、ひとつのハッシュの中に同一の値を持つ要素が 2 個以上含まれていてもかまいません。

ハッシュを構成している要素の個数を、そのハッシュの「大きさ」(size) と呼びます。そして、大きさが 0 のハッシュは、「空ハッシュ」(empty hash) と呼ばれます。

4.3.2　ハッシュを生成する方法

ハッシュは、配列と同じように、要素を列挙することによって生成することができます。

要素を列挙することによってハッシュを生成する場合、ハッシュのひとつの要素は、

式 $_1$ => 式 $_2$

という形の記述によってあらわされます。このような記述を書くと、式 $_1$ の値が要素のキーになって、式 $_2$ の値が要素の値になります。たとえば、

```
"ichigo"=>180
```

と書くことによって、キーが `ichigo` という文字列で値が 180 という整数であるような要素をあらわすことができます。

要素を列挙することによってハッシュを生成したいときは、

{ 要素 , 要素 , 要素 , ··· }

という形の式を書きます。つまり、ハッシュのそれぞれの要素をコンマ (`,`) で区切って並べて、その全体を中括弧 (`{}`) で囲むわけです。この形の式を評価すると、その中に記述された要素から構成されるハッシュが値として得られます。たとえば、

```
{"momo"=>180, "kaki"=>140, "nashi"=>200}
```

という式を評価すると、キーが `momo` で値が 180 という要素、キーが `kaki` で値が 140 という要素、キーが `nashi` で値が 200 という要素、という 3 個の要素から構成されるハッシュが、値として得られます。

ひとつのハッシュの中で、キーのクラスや値のクラスを一定にしないといけないという制約はありません。ですから、

```
{"iruka"=>63, 707=>true, false=>"kujira"}
```

というような、キーや値のクラスが混在しているハッシュを作ることも可能です。

中括弧だけの式、つまり `{}` という式を評価すると、値として空ハッシュが得られます。

---

[2] ハッシュは、「辞書」(dictionary) とか「連想配列」(associative array) などと呼ばれることもあります。

メッセージ式	説明
h.size	h の大きさを返します。
h.keys	h のすべてのキーから構成される配列を返します。
h.values	h のすべての値から構成される配列を返します。
h.to_a	h の要素をあらわす配列から構成される配列を返します。
h.empty?	h が空ハッシュならば true、そうでなければ false を返します。
h.key?(k)	h がキー k を持つならば true、そうでなければ false を返します。
h.value?(v)	h が値 v を持つならば true、そうでなければ false を返します。

表 4.3: ハッシュの基本的なメソッド

4.3.3 ハッシュの基本的なメソッド

ハッシュは、自分自身を取り扱うためのさまざまなメソッドを持っています。表 4.3 は、ハッシュが持っているメソッドのうちの基本的なものを示しています。

keys と values と to_a は、いずれも、レシーバーを配列に変換するメソッドです。keys は、要素のキーだけから構成される配列を返すメソッドで、values は、要素の値だけから構成される配列を返すメソッドです。to_a は、それぞれの要素を、キーと値から構成される配列に変換して、それらの配列から構成される配列を返します。

```
irb(main):001:0> h = {"a"=>37, "b"=>80, "c"=>21, "d"=>14}
=> {"a"=>37, "b"=>80, "c"=>21, "d"=>14}
irb(main):002:0> h.keys
=> ["a", "b", "c", "d"]
irb(main):003:0> h.values
=> [37, 80, 21, 14]
irb(main):004:0> h.to_a
=> [["a", 37], ["b", 80], ["c", 21], ["d", 14]]
```

4.3.4 ハッシュからの要素の取り出し

ハッシュから要素を取り出したいときは、ハッシュが持っている [] という演算を使います。h がハッシュで、k がオブジェクトだとするとき、

h[k]

という式で [] を呼び出すと、[] は、k をキーとする要素を h から取り出して、その要素の値を戻り値として返します。

```
irb(main):001:0> tensuu = {"sansuu"=>22, "rika"=>61}
=> {"sansuu"=>22, "rika"=>61}
irb(main):002:0> tensuu["rika"]
=> 61
```

4.3.5 ハッシュの要素の追加と値の変更

ハッシュに要素を追加したいとき、または要素の値を変更したいときは、ハッシュが持っている []= という演算を使います。h がハッシュで、k と v がオブジェクトだとするとき、

h[k] = v

という式で []= を呼び出すと、[]= は、k をキーとする h の要素が存在していなかった場合は、キーが k で値が v という要素を h に追加します。k をキーとする h の要素がすでに存在していた場合は、その要素の値を v に変更します。

```
irb(main):001:0> tensuu = {"sansuu"=>22, "rika"=>61}
=> {"sansuu"=>22, "rika"=>61}
irb(main):002:0> tensuu["kokugo"] = 93
=> 93
irb(main):003:0> tensuu
=> {"kokugo"=>93, "sansuu"=>22, "rika"=>61}
irb(main):004:0> tensuu["sansuu"] = 87
=> 87
```

```
irb(main):005:0> tensuu
=> {"kokugo"=>93, "sansuu"=>87, "rika"=>61}
```

次のプログラムの中で定義されている toHash という配列のメソッドは、引数として配列を受け取って、レシーバーを構成するそれぞれの要素をキー、引数を構成するそれぞれの要素を値とするハッシュを戻り値として返します。

プログラムの例　tohash.rb

```
class Array
  def toHash(v)
    h = {}
    size.times { |i| h[self[i]] = v[i] }
    h
  end
end
```

実行例

```
irb(main):001:0> load("tohash.rb")
=> true
irb(main):002:0> ["a", "b", "c", "d"].toHash([34, 27, 52, 93])
=> {"a"=>34, "b"=>27, "c"=>52, "d"=>93}
```

4.3.6　ハッシュの要素の削除

ハッシュの要素を削除したいときは、delete というメソッドを使います。このメソッドは、引数として受け取ったオブジェクトをキーとする要素をレシーバーから削除して、削除した要素の値を戻り値として返します。

```
irb(main):001:0> tensuu = {"sansuu"=>22, "rika"=>61}
=> {"sansuu"=>22, "rika"=>61}
irb(main):002:0> tensuu.delete("rika")
=> 61
irb(main):003:0> tensuu
=> {"sansuu"=>22}
```

4.3.7　デフォルト値

ハッシュが持っている [] という演算は、指定されたキーを持つ要素がレシーバーの中に存在していなかった場合、「デフォルト値」(default value) と呼ばれるオブジェクトを返します。

デフォルト値は、それが明示的に設定されていなければ、nil が設定されています。

```
irb(main):001:0> tensuu = {"kokugo"=>93, "sansuu"=>87}
=> {"kokugo"=>93, "sansuu"=>87}
irb(main):002:0> tensuu["rika"]
=> nil
```

ハッシュに設定されているデフォルト値を変更したいときは、default= というメソッドを使います。h がハッシュで d がオブジェクトだとするとき、

h.default = d

という式で default= を呼び出すと、h のデフォルト値が d に変更されます。

```
irb(main):001:0> nedan = {"mikan"=>220, "ringo"=>180}
=> {"ringo"=>180, "mikan"=>220}
irb(main):002:0> nedan["ichigo"]
=> nil
irb(main):003:0> nedan.default = 200
=> 200
irb(main):004:0> nedan["ichigo"]
=> 200
```

4.3.8　ハッシュのイテレーター

ハッシュは、each というイテレーターを持っています。このイテレーターは、レシーバーのそれぞれの要素について、そのキーと値を渡してブロックを実行します。

次のプログラムの中で定義されている intersection というハッシュのメソッドは、引数とし

メッセージ式	説明
i.read	i の内容を読み込んで文字列として返します。
i.readlines	i の内容を読み込んで行の配列として返します。
i.gets	i から 1 行を読み込んで返します。
i.each	行単位で読み込みを繰り返すイテレーターです。
i.getc	i から 1 文字を読み込んで文字コードとして返します。
i.each_byte	文字単位で読み込みを繰り返すイテレーターです。
i.write(s)	i に文字列 s を書き込みます。

表 4.4: 読み込みと書き込みのメソッド

て 1 個のハッシュを受け取って、レシーバーと引数とで共通するキーを持つ要素を統合することによってできるハッシュを戻り値として返します。

プログラムの例 intsec.rb

```
class Hash
  def intersection(h)
    intsec = {}
    each do |k, v|
      if h.key?(k)
        intsec[k] = [v, h[k]]
      end
    end
    intsec
  end
end
```

実行例

```
irb(main):001:0> load("intsec.rb")
=> true
irb(main):002:0> h1 = {"kuma"=>37, "tanuki"=>21, "shika"=>85}
=> {"shika"=>85, "tanuki"=>21, "kuma"=>37}
irb(main):003:0> h2 = {"tanuki"=>78, "mogura"=>44, "kuma"=>62}
=> {"kuma"=>62, "tanuki"=>78, "mogura"=>44}
irb(main):004:0> h1.intersection(h2)
=> {"kuma"=>[37, 62], "tanuki"=>[21, 78]}
```

4.3.9 中括弧の省略

メソッドに渡す引数として、要素を列挙してハッシュを生成する式を書く場合、引数がそれ 1 個だけならば、その式の中括弧は省略することができます。

それでは、irb を使って実際に確かめてみましょう。まず、

```
def sonomama(a)
  a
end
```

という式を入力してください。そうすると、受け取った引数をそのまま返すだけの、sonomama という関数的メソッドが定義されます。次に、

```
sonomama("taika"=>645, "meiji"=>1868, "heisei"=>1989)
```

という式を入力してください。すると、1 個のハッシュが戻り値として返ってきます。つまり、メッセージ式の丸括弧の中にハッシュの要素を列挙すると、それらの要素から構成される 1 個のハッシュを引数として渡すという意味になるわけです。

文字列	説明
r	ファイルの先頭から読み込む。デフォルト。
w	ファイルの内容を削除したのちに書き込む。
a	ファイルの内容を削除しないで、その末尾に書き込む。
r+	ファイルの先頭から読み書きをする。
w+	ファイルの内容を削除したのちに読み書きをする。
a+	ファイルの末尾から読み書きをする。
b	上記の文字列の右側に書いて、バイナリーモードを指定する。

表 4.5: オープンモードをあらわす文字列

4.4　ファイル

4.4.1　ファイルの基礎

Ruby には、IO という組み込みクラスがあります。このクラスは、プログラムがデータを読み込んだり書き込んだりする対象をあらわすオブジェクトを生成します。IO またはそれを継承するクラスのインスタンスは、「IO オブジェクト」(IO object) と呼ばれます。IO オブジェクトは、データを書き込んだり読み込んだりするためのさまざまなメソッドを持っています。表 4.4 は、IO オブジェクトが持っている読み込みと書き込みのメソッドを示しています。

ファイルは、プログラムがデータの読み込みや書き込みをする対象のひとつです。ですから、ファイルをあらわす IO オブジェクトを生成して、それが持っているメソッドを使うことによって、そのファイルに対してデータの読み書きを実行することができます。

ファイルに対する読み書きをするための IO オブジェクトは、IO クラスをスーパークラスとする File という組み込みクラスから生成されます。

ファイルからデータを読み込んだり、ファイルにデータを書き込んだりするためには、それに先立って、そのための準備をする必要があります。ファイルに対する読み書きのための準備をすることを、ファイルを「オープンする」(open) と言います。

ファイルに対して読み書きを実行したときは、それが終わったのち、その後始末をする必要があります。ファイルに対する読み書きが終わったのちにその後始末をすることを、ファイルを「クローズする」(close) と言います。

4.4.2　ファイルをあらわす IO オブジェクトの生成

ファイルをあらわす IO オブジェクトを生成したいときは、open という関数的メソッドを使います。

open は、二つの文字列を引数として受け取ります。ひとつ目はファイルを指定するパス名で、二つ目は「オープンモード」(open mode) と呼ばれるものをあらわす文字列です。オープンモードというのは、ファイルをオープンする目的のことです。表 4.5 は、オープンモードをあらわす文字列の意味を示しています。たとえば、データを読み込むためにファイルをオープンしたいときは、r という文字列を二つ目の引数として open に渡します。

open は、引数で指定されたファイルを指定されたオープンモードでオープンして、そのファイルをあらわす IO オブジェクト（File クラスのインスタンス）を渡してブロックを実行します。そして、ブロックの実行が終了したのち、そのファイルをクローズします。ファイルは、ブロックの実行が正常に終了した場合だけではなくて、例外が発生して実行が中断した場合も、かならずクローズされます。

たとえば、

```
open("namako.txt", "r") ブロック
```

というような式で open を呼び出したとすると、open は、namako.txt というパス名のファイルを、そこからデータを読み込むという目的でオープンして、そのファイルをあらわす IO オブジェクトを渡してブロックを実行します。

なお、open に渡す二つ目の引数、つまりオープンモードをあらわす文字列は、読み込みのためにファイルをオープンする場合は省略することができます。ですから、上の式は、

```
open("namako.txt") ブロック
```

と書いても同じ意味になります。

なお、openは、オープンモードがw、a、w+、a+のいずれかで、存在しないファイルのパス名が指定された場合は、新しい空のファイルを作って、そのファイルをあらわすIOオブジェクトを生成します。

4.4.3 読み書き位置

IOオブジェクトがあらわしているものに対する読み書きは、「読み書き位置」(read/write position) と呼ばれる位置に対して実行されます。

rでファイルをオープンした場合、オープンした直後の読み書き位置は、ファイルの先頭です。

wでファイルをオープンした場合も、ファイルは空になっているわけですから、最初の読み書き位置はファイルの先頭です。それに対して、aでファイルをオープンした場合、最初の読み書き位置はファイルの末尾になります。

データの読み書きをするメソッドは、読み書き位置に対して読み書きを実行して、そののち、次に読み書きを実行するべき位置へ読み書き位置を移動させます。

ファイルの末尾という位置のことを「ファイルの終わり」(end of file) と呼びます。

4.4.4 ファイル全体の読み込み

Fileクラスのインスタンスは、IOクラスから継承した、データの読み込みや書き込みのためのさまざまなメソッドを持っていますので、それらのメソッドを使うことによって、ファイルに対する読み込みや書き込みを実行することができます。

ファイルの中のデータをすべて読み込みたいときは、readというメソッドを使います。readは、読み書き位置からファイルの終わりまでのデータを読み込んで、それをひとつの文字列として返します。

次のプログラムは、ファイルの内容をそのまま出力します。

プログラムの例 read.rb

```
if ARGV.size == 1
  open(ARGV[0]) do |f|
    f.read.display
  end
else
  "使い方: ruby read.rb パス名\n".display
end
```

ファイルの内容をすべて読み込むメソッドとしては、readのほかに、readlinesというのもあります。readlinesは、ファイルの内容を行に分割して、それぞれの行から構成される配列を戻り値として返します。なお、配列を構成するそれぞれの行の末尾には、改行をあらわす文字が付いたままになっています。文字列の末尾にある改行は、文字列が持っているchompというメソッドを使うことによって取り除くことができます。

次のプログラムは、ファイルを構成するそれぞれの行を角括弧で囲んで出力します。

プログラムの例 lines.rb

```
if ARGV.size == 1
  open(ARGV[0]) do |f|
    f.readlines.each do |line|
      ("[" + line.chomp + "]\n").display
    end
  end
else
  "使い方: ruby lines.rb パス名\n".display
end
```

4.4.5 行単位での読み込み

ファイルからデータを1行だけ読み込みたいときは、getsというメソッドを使います。getsは、読み書き位置からデータを1行だけ読み込んで、それを戻り値として返します（改行も含ま

れます)。ただし、読み書き位置がファイルの終わりに到達している場合は、戻り値として nil を返します。

次のプログラムは、ファイルを構成するそれぞれの行に番号を付けて出力します。

プログラムの例　linenum.rb

```
if ARGV.size == 1
  open(ARGV[0]) do |f|
    n = 0
    while line = f.gets
      n += 1
      (n.to_s + ": " + line).display
    end
  end
else
  "使い方: ruby linenum.rb パス名\n".display
end
```

ファイルの終わりに到達するまでファイルからの読み込みを繰り返すという動作を記述するときは、このプログラムのように、 while 式がしばしば使われます。

while 式を使って読み込みを繰り返す場合には、繰り返しの条件のところに読み込みを実行する式を書く、というのが常套手段です。読み込みを実行するメソッドの多くは、ファイルの終わりに到達すると nil（つまり偽）を返すように定義されていますので、そのように書いておくことによって、ファイルの終わりに到達したところで繰り返しを終了させることができます。

4.4.6　行単位での読み込みを繰り返すイテレーター

ファイルからデータを行単位で読み込むという動作を繰り返すための方法としては、 gets と while 式を使う方法のほかに、 each というイテレーターを使うという方法もあります。

each は、読み込んだ行（改行も含まれます）を渡してブロックを実行するということを、ファイルの終わりに到達するまで繰り返すイテレーターです。

次のプログラムは、ファイルの内容を出力したのち、それを構成している行の個数を出力します。

プログラムの例　count.rb

```
if ARGV.size == 1
  open(ARGV[0]) do |f|
    count = 0
    f.each do |line|
      count += 1
      line.display
    end
    (count.to_s + " lines\n").display
  end
else
  "使い方: ruby count.rb パス名\n".display
end
```

4.4.7　文字単位での読み込み

ファイルからデータを 1 文字だけ読み込みたいときは、 getc というメソッドを使います。 getc は、読み書き位置からデータを 1 文字だけ読み込んで、その文字の文字コードを戻り値として返します。ただし、読み書き位置がファイルの終わりに到達している場合は、戻り値として nil を返します。

文字コードを文字列に変換したいときは、整数が持っている chr というメソッドを使います。 chr は、レシーバーを文字コードとする文字から構成される、長さが 1 の文字列を返すメソッドです。たとえば、イコール (=) という文字の文字コードは 61 ですから、

```
61.chr
```

という式を評価すると、 "=" という文字列が値として得られます。

次のプログラムは、ファイルの内容に含まれているすべての空白をドット (.) に変換した結果を出力します。

プログラムの例 sptodot.rb

```
if ARGV.size == 1
  open(ARGV[0]) do |f|
    while c = f.getc
      if c == 32   # 32 は空白の文字コード
        ".".display
      else
        c.chr.display
      end
    end
  end
else
  "使い方: ruby sptodot.rb パス名\n".display
end
```

4.4.8 文字単位での読み込みを繰り返すイテレーター

ファイルからデータを文字単位で読み込むという動作を繰り返すための方法としては、getc と while 式を使う方法のほかに、each_byte というイテレーターを使うという方法もあります。

each_byte は、読み込んだ文字の文字コードを渡してブロックを実行するということを、ファイルの終わりに到達するまで繰り返すイテレーターです。

次のプログラムは、コマンドライン引数で指定された文字数で、ファイルを構成する行を再編成して、その結果を出力します。

プログラムの例 fixline.rb

```
if ARGV.size == 2
  n = ARGV[1].to_i
  open(ARGV[0]) do |f|
    count = 0
    f.each_byte do |c|
      if c != 10   # 10 は改行の文字コード
        count += 1
        if count == n
          "\n".display
          count = 0
        end
        c.chr.display
      end
    end
  end
else
  "使い方: ruby fixline.rb パス名 1 行の文字数\n".display
end
```

4.4.9 書き込み

ファイルにデータを書き込みたいときは、write というメソッドを使います。write は、引数として文字列を受け取って、読み書き位置にその文字列を書き込みます。

次のプログラムは、ファイルの内容を別のファイルにコピーします。

プログラムの例 copy.rb

```
if ARGV.size == 2
  open(ARGV[0]) do |inf|
    open(ARGV[1], "w") do |outf|
      outf.write(inf.read)
    end
  end
else
  "使い方: ruby copy.rb コピー元 コピー先\n".display
end
```

4.4.10　標準入出力

オペレーティングシステムは、「標準入出力」(standard IO) と呼ばれる三つのファイルを持っています。標準入出力は、実体を持たない仮想的なファイルで、オペレーティングシステムによって特定のファイルに割り当てられることによって、実際に読み書きをすることが可能になります。

標準入出力を構成する三つのファイルは、それぞれ、「標準入力」(standard input)、「標準出力」(standard output) 「標準エラー」(standard error) と呼ばれます。通常、標準入力はキーボードに、標準出力と標準エラーはモニターに割り当てられていますが、それらの割り当てを別のファイルに切り替えることも可能です。標準入出力をキーボードやモニターから別のファイルに切り替えることを、標準入出力を「リダイレクトする」(redirect) と言います（名詞は「リダイレクション」(redirection) です)。

ちなみに、`display` というのは、レシーバーを標準出力に書き込むメソッドです。

Ruby では、標準入出力をあらわす IO オブジェクトのそれぞれに、

`STDIN`	標準入力
`STDOUT`	標準出力
`STDERR`	標準エラー

という名前が与えられています。ですから、

```
line = STDIN.gets
```

という式を評価することによって、標準入力（通常はキーボード）から 1 個の行を読み込んで、それを `line` という変数に代入することができます。

次の `irblike.rb` は、irb と同じように、ruby を対話的に使うことができるようにするプログラムです（irb に比べると、機能がかなり貧弱ですが)。

プログラムの例　`irblike.rb`

```
while true
  "irblike> ".display
  begin
    line = STDIN.gets
    ("=> " + eval(line).inspect + "\n").display
  rescue => e
    (e.class.to_s + ": " + e.message + "\n" +
    e.backtrace.join("\n") + "\n").display
  end
end
```

このプログラムは、irb と同じように、式として `exit` を入力することによって終了させることができます。

ところで、このプログラムには、ここで初めてこの文章に登場するメソッドが、三つ、使われていますので、それらを簡単に紹介しておくことにしましょう。

`eval`	関数的メソッドです。引数として文字列を受け取って、それを `Ruby` の式とみなして評価して、その結果として得られた値を戻り値として返します。
`inspect`	すべてのオブジェクトが持っているメソッドです。`to_s` と同じように、レシーバーを文字列に変換した結果を返します。`inspect` と `to_s` との相違点は、前者のほうが後者よりも、レシーバーを想起させやすい文字列に変換するという点にあります。
`backtrace`	例外が持っているメソッドです。レシーバーが発生した時点で実行の途上にあるメソッドについての情報を、文字列の配列という形式で返します。

4.4.11　フィルター

標準入力からデータを読み込んで、そのデータに対して何らかの処理を実行して、その結果を標準出力に書き込む、という動作をするプログラムのことを、「フィルター」(filter) と呼びます。ただし、ほとんどのフィルターは、コマンドライン引数でパス名が指定された場合は、標準入力ではなくて指定されたファイルからデータを読み込むように作られています。

たとえば、UNIX の cat というプログラムもフィルターのひとつです。コマンドライン引数を何も渡さないで cat を起動すると、cat は、標準入力からデータを読み込んで、それをそのまま標準出力に書き込みます。コマンドライン引数を何個か渡して起動した場合は、引数の順番のと

おりに、引数によって指定されたファイルからデータを読み込んで、それを標準出力に書き込みます。

4.4.12 リダイレクション

標準入出力は、通常はキーボードやモニターに割り当てられているわけですが、それをファイルに切り替える（つまりリダイレクトする）ということも可能です。

標準入力をキーボードからファイルにリダイレクトしたいときは、シェルに対して、

コマンド < パス名

という形のコマンドを入力します。そうすると、標準入力は小なり (<) の右側のパス名で指定されたファイルにリダイレクトされて、その状態で、小なりの左側のコマンドが実行されます。たとえば、

```
cat < asagao.txt
```

というコマンドをシェルに入力すると、asagao.txt の内容がモニターに出力されます。

標準出力をモニターからファイルにリダイレクトしたいときは、シェルに対して、

コマンド > パス名

という形のコマンドを入力します。そうすると、標準出力は大なり (>) の右側のパス名で指定されたファイルにリダイレクトされて、その状態で、大きいなりの左側のコマンドが実行されます。たとえば、

```
cat > himawari.txt
```

というコマンドをシェルに入力すると、キーボードに入力したデータが himawari.txt に書き込まれます。

大なり (>) ではなくて、大なり大なり (>>) を使って標準出力をリダイレクトすることも可能です。大なり大なりを使った場合、標準出力に書き込んだデータは、ファイルの末尾に追加されます。

4.4.13 パイプ

標準入出力はファイルにリダイレクトすることができるわけですが、それだけではなくて、標準入出力を媒介にして二つのプログラムを結合する、ということも可能です。それをしたいときは、「パイプ」(pipe) と呼ばれる機構を使います。

シェルのコマンドの中では、パイプは、縦棒 (|) という文字によってあらわされます。シェルに対して、

コマンド$_1$ | コマンド$_2$

という形のコマンドを入力すると、コマンド$_1$で起動されたプログラムの標準出力と、コマンド$_2$で起動されたプログラムの標準入力とが結合されます。たとえば、

```
ls -l | cat -n
```

というコマンドをシェルに入力すると、ディレクトリの一覧に行番号を付けたものが出力されます。

4.4.14 ARGF

スクリプト言語の多くは、フィルターが簡単に書けるようにするための機能を持っています[3]。Ruby もその例外ではありません。

Ruby では、フィルターを簡単に書くことができるようにするために、ARGF という名前のものが準備されています。ARGF というのは、コマンドライン引数をパス名とみなして、それらのパス名で指定されたファイルを、その順番のとおりに連結してできた仮想的なファイル、というものをあらわしているオブジェクトです。そして、コマンドライン引数が 0 個の場合、ARGF は、標準入力をあらわすオブジェクトになります。

ARGF があらわしているファイルは、読み込みのオープンモードでオープンされています。ARGF があらわしているファイルからデータを読み込みたいときは、それが持っているメソッドを使い

[3] sed や awk のように、フィルターを書くことを第一の目的として設計されたスクリプト言語もあります。

ます。ARGF が持っている読み込みのメソッドは、IO オブジェクトが持っているものと、名前も動作も同じです。

それでは、実際にフィルターを書いてみましょう。次のプログラムは、大なり (>) と空白を行の先頭に挿入するフィルターです。

プログラムの例　quote.rb

```
ARGF.each do |line|
  ("> " + line).display
end
```

このプログラムを、たとえば、

```
ruby quote.rb namako.txt umiushi.txt hitode.txt
```

というコマンドで起動したとすると、このプログラムは、コマンドライン引数で指定された 3 個のファイルを、その順番のとおりに連結することによってできた仮想的なファイルの内容を読み込んで、大なりと空白をそれぞれの行の先頭に挿入した結果を出力します。

コマンドライン引数が 0 個の場合、ARGF は標準入力を扱うことになりますので、

```
cal 7 1999 | ruby quote.rb
```

というコマンドをシェルに入力したとすると、quote.rb は、cal が出力したカレンダーを処理することになります。

4.4.15　ARGF からファイルを除外する方法

フィルターに渡すコマンドライン引数は、かならずしもすべてが読み込みの対象となるファイルのパス名とは限りません。しかし、Ruby の処理系は、ARGF を作るとき、コマンドライン引数はすべてパス名だと頭ごなしに決め付けてしまいます。それでは、読み込むファイルのパス名ではないコマンドライン引数を受け取るフィルターは、いったいどのように書けばいいのでしょうか。

Ruby のプログラムは、コマンドライン引数を ARGV という名前の配列として受け取ります。たとえば、

```
ruby tanpopo.rb sumire renge nanohana
```

というコマンドで tanpopo.rb というプログラムを起動したとすると、そのプログラムの中の ARGV は、

```
["sumire", "renge", "nanohana"]
```

という配列になります。

実は、ARGF の対象となるパス名というのは、ARGV の内容によって決定されるのです。ですから、ARGV から要素を削除すれば、その要素は ARGF の対象から除外されますし、ARGV に要素を追加すれば、その要素は ARGF の対象として認識されます。

配列の先頭の要素を削除したいときは、配列が持っている shift というメソッドを使います。shift は、レシーバーの先頭の要素を削除して、その要素を戻り値として返します（レシーバー自体が変化するという点に注意してください)。たとえば、

```
a = [24, 18, 60, 77]
```

という式で配列を変数に代入して、そののち、

```
a.shift
```

という式を評価したとすると、式の値として 24 が得られて、変数の中の配列は、

```
[18, 60, 77]
```

に変化します。

次のプログラムは、1 個目のコマンドライン引数と大なり (>) と空白を行の先頭に挿入するフィルターです。

プログラムの例　quote2.rb

```
name = ARGV.shift
ARGF.each do |line|
  (name + "> " + line).display
end
```

メッセージ	説明
delete(p)	パス名 p を持つものを削除します。
rename(p, q)	パス名を p から q へ変更します。
size(p)	パス名 p を持つファイルの大きさを返します。
mtime(p)	パス名 p を持つファイルの最終更新時刻を返します。
exist?(p)	パス名 p を持つものが存在するならば true を返します。
file?(p)	パス名 p を持つものがファイルならば true を返します。
directory?(p)	パス名 p を持つものがディレクトリならば true を返します。
basename(p)	パス名 p の末尾の名前を返します。
basename(p, e)	パス名 p の末尾の名前から拡張子 e を取り除いた残りを返します。
dirname(p)	パス名 p から末尾の名前を取り除いた残りを返します。
split(p)	パス名 p を末尾とそれ以外に分解して、配列にして返します。
join(s, ···)	パス名の区切り文字をはさんで引数を連結した結果を返します。
expand_path(p)	パス名 p を絶対パス名に変換した結果を返します。

表 4.6: File のクラスメソッド

4.4.16 オプション

コマンドライン引数のうちで、プログラムに対して通常とは異なる動作を指示するもののことを、「オプション」(option) と呼びます。オプションは、普通、オプションではない普通の引数と区別するために、先頭にマイナス (-) という文字を書きます。

オプションを受け付けるフィルターを作るためには、コマンドライン引数がオプションかどうかを判断して、オプションならばそれを ARGV から削除するという処理を書く必要があります。

次のプログラムは、ファイルの内容をそのまま出力するフィルターですが、-n というオプションを付けて起動すると、それぞれの行の先頭に行番号を付けます。

プログラムの例 cat.rb

```
if ARGV[0] == "-n"
  ARGV.shift
  n = 0
  ARGF.each do |line|
    n += 1
    (n.to_s + ": " + line).display
  end
else
  ARGF.read.display
end
```

4.4.17 File のクラスメソッド

File というクラスは、クラスメソッドとして、ファイルやディレクトリを取り扱うためのさまざまなメソッドを持っています。表 4.6 は、File クラスが持っているクラスメソッドのうちの主要なものを示しています。

4.5 ディレクトリ

4.5.1 Dir のクラスメソッド

Ruby のインタプリタには、Dir という名前のクラスが組み込まれています。このクラスは、クラスメソッドとして、ディレクトリを取り扱うためのさまざまなメソッドを持っています。表 4.7 は、Dir クラスが持っているクラスメソッドのうちの主要なものを示しています。

メッセージ	説明
pwd	カレントディレクトリの絶対パス名を返します。
chdir(p)	パス名 p を持つディレクトリをカレントディレクトリにします。
mkdir(p)	パス名 p を持つディレクトリを作ります。
rmdir(p)	パス名 p を持つディレクトリを削除します。
entries(p)	パス名 p を持つディレクトリの中にあるものの名前の配列を返します。

表 4.7: Dir のクラスメソッド

4.5.2　ディレクトリの内容に対する処理

ディレクトリの中にあるすべてのものに対して何らかの処理を実行したい、というときは、まず、ディレクトリの中にあるものの名前を調べて、次に、それらの名前を持つものに対する処理を繰り返します。

ディレクトリの中にあるものの名前を調べたいときは、Dir クラスが持っている entries というクラスメソッドを使います。このメソッドは、引数で指定されたディレクトリの中にあるものの名前から構成される配列を戻り値として返します。なお、その配列は、指定されたディレクトリ自身 (.) とその親ディレクトリ (..) も含んでいる、という点に注意してください。

次のプログラムは、指定されたディレクトリの中にあるもののそれぞれについて、それがディレクトリならば、その名前を角括弧で囲んだものを出力して、それがファイルならば、その名前と大きさと最終更新時刻を出力します。

プログラムの例　dirlist.rb

```
class String
  def joinPath(p)
    if self[size - 1, 1] == "/"
      self + p
    else
      self + "/" + p
    end
  end
end

def dirlist(p)
  Dir.entries(p).sort.each do |e|
    pe = p.joinPath(e)
    if File.directory?(pe)
      ("[" + e + "]\n").display
    elsif File.file?(pe)
      ([e, File.size(pe), File.mtime(pe)].join(", ") +
      "\n").display
    end
  end
end

if ARGV.size == 1
  dirlist(ARGV[0])
else
  "使い方: ruby dirlist.rb パス名\n".display
end
```

4.5.3　再帰的なディレクトリの処理

メソッドを再帰的に定義することによって、指定されたディレクトリだけではなくて、そのディレクトリを根とする木の全体を処理することができます。

次のプログラムは、指定されたディレクトリを根とする木の中にあるすべてのディレクトリについて、その大きさ（その中にあるファイルの大きさの合計）を出力します。

プログラムの例　dirsize.rb

```
class String
```

```
  def joinPath(p)
    if self[size - 1, 1] == "/"
      self + p
    else
      self + "/" + p
    end
  end
end

def dirsize(p)
  sum = 0
  (Dir.entries(p) - [".", ".."]).each do |e|
    pe = p.joinPath(e)
    if File.directory?(pe)
      sum += dirsize(pe)
    elsif File.file?(pe)
      sum += File.size(pe)
    end
  end
  (p + ": " + sum.to_s + "\n").display
  sum
end

if ARGV.size == 1
  dirsize(ARGV[0])
else
  "使い方: ruby dirsize.rb パス名\n".display
end
```

4.6 正規表現

4.6.1 正規表現とは何か

文字列を処理するときには、しばしば、その中に含まれている特定のパターン（構造）を探し出す必要が生じます。そして、パターンを探し出すという処理を記述するためには、そのパターンそのものを記述する方法が必要になります。文字列のパターンを記述する方法としては、「正規表現」と呼ばれるものがもっともよく使われています。

「正規表現」(regular expression) というのは、文字列を使って文字列のパターンを表現するための言語のひとつです。また、この言語を使って作られた文字列のことも、「正規表現」と呼ばれます。

正規表現によってあらわされるパターンと文字列とが一致することを、正規表現（またはそれがあらわしているパターン）と文字列とが「マッチする」(match) と言います（名詞は「マッチング」(matching) です）。

文字列を扱うプログラムの多くは、文字列のパターンを表現するために正規表現を使っています。その代表的な例として、grep というプログラムがあります。grep はフィルターの一種で、コマンドライン引数として正規表現を受け取って、読み込んだ文字列のうちで、受け取った正規表現とマッチする文字列が含まれている行だけを出力する、という動作をします。

テキストエディターも、たいていのものは正規表現が扱えるように作られています。また、sed、awk、Perl、Ruby のような、スクリプト言語と呼ばれる言語の大多数は、その一部分として正規表現を含んでいます。

4.6.2 正規表現の基礎の基礎

1 個の特定の文字というパターンをあらわす正規表現は、たいていの場合、その文字そのものです。たとえば、数字の `8` という文字は、数字の `8` という正規表現によってあらわされます。

特定のパターンのうしろに特定のパターンが続いているという構造のことを「連接」(sequence) と呼びます。連接は、正規表現と正規表現とをパターンの順番のとおりに並べることによって表現することができます。たとえば、`pi` という正規表現は、`p` という文字のうしろに `i` という文字が続いているというパターン、つまり `pi` という文字列をあらわします。

ですから、たいていの場合、特定の文字列というパターンをあらわす正規表現は、それとまったく同じ文字列になります。たとえば、`kamome` という文字列は、`kamome` という正規表現によっ

てあらわされます。

4.6.3　正規表現オブジェクト

Ruby では、正規表現は、Regexp というクラスのインスタンスとして扱われます。このクラスのインスタンスは、「正規表現オブジェクト」(regular expression object) と呼ばれます。

Regexp クラスのインスタンスを作る方法は二つあります。ひとつは「正規表現リテラル」(regular expression literal) と呼ばれるリテラルを書く方法で、もうひとつは Regexp クラスが持っている new というクラスメソッドを使う方法です。

正規表現リテラルは、

/ 正規表現 /

というように、正規表現の前後にスラッシュを書くことによって作ります。正規表現リテラルを評価すると、その正規表現を扱う正規表現オブジェクトが値として得られます。たとえば、

```
re = /suzume/
```

という式を評価すると、suzume という正規表現を扱う正規表現オブジェクトが作られて、それが re という変数に代入されます。

Regexp クラスの new というクラスメソッドは、文字列を正規表現オブジェクトに変換したいときに使われます。このメソッドは、引数として 1 個の文字列を受け取って、その文字列を正規表現オブジェクトに変換して、それを戻り値として返します。たとえば、

```
re = Regexp.new("uguisu")
```

という式を評価すると、uguisu という正規表現を扱う正規表現オブジェクトが作られて、それが re という変数に代入されます。

4.6.4　マッチングの演算

文字列の中に含まれている特定のパターンを探し出したいときは、正規表現オブジェクトが持っている =~ という演算を使います。この演算は、引数として文字列を受け取って、レシーバーとマッチする部分文字列を引数の中で探索します。そして、マッチする部分文字列のうちで、もっとも左にあるものの位置をあらわす整数（先頭の文字を 0 番目と数えます）を戻り値として返します。たとえば、

```
/game/ =~ "oyagamekogame"
```

という式を評価したとすると、値として 3 という整数が得られます。

レシーバーとマッチする部分文字列が引数の中にまったく存在しない場合、=~ は、戻り値として nil を返します。=~ の戻り値を真偽値と解釈すると、整数は真という意味で、nil は偽という意味になります。ですから、=~ は、文字列の中に正規表現とマッチする部分文字列が含まれているかどうかを調べる述語だと考えることもできます。つまり、

```
if 正規表現 =~ 文字列
  マッチした場合の処理
else
  マッチしなかった場合の処理
end
```

という形の if 式を書くことによって、特定のパターンとマッチする部分文字列が文字列の中に含まれているかどうかで動作を選択することができる、ということです。

次のプログラムは、grep とほぼ同じ動作をするフィルターを Ruby で書いてみたものです。

プログラムの例　grep.rb

```
re = Regexp.new(ARGV.shift)
ARGF.each do |line|
  if re =~ line
    line.display
  end
end
```

実行例

```
$ ruby grep.rb line grep.rb
ARGF.each do |line|
  if re =~ line
    line.display
```

正規表現オブジェクトだけではなくて、文字列のオブジェクトも、文字列の中に含まれている特定のパターンを探し出す、 =~ という演算を持っています。

正規表現オブジェクトの =~ と文字列のオブジェクトの =~ は、レシーバーと引数が逆になっているという点が違うだけで、それ以外の動作は同じです。ですから、

```
"oyagamekogame" =~ /game/
```

という式を評価したとすると、値として 3 という整数が得られます。

4.6.5 メタ文字

正規表現を構成するそれぞれの文字は、基本的には、自分自身というパターンを意味しています。しかし、特定の文字列ではなくて、いくつかの文字列とマッチするようなパターンを表現するためには、自分自身ではない特別な意味をいくつかの文字に与える必要があります。

正規表現を書くために使われる、自分自身ではなくて何か別のことを意味している文字は、「メタ文字」(metacharacter) と呼ばれます。

どの文字をメタ文字として使うのかというのは、正規表現を扱うプログラムごとに多少の違いがありますが、それらのプログラムのうちの多くは、

```
\ . [ ] - ^ $ * + ? { } ( ) |
```

というような文字をメタ文字として使っていて、それぞれのメタ文字の意味も、ほぼ統一されています。

4.6.6 エスケープ

ところで、メタ文字自身をあらわす正規表現、つまり特定のメタ文字だけとマッチする正規表現は、どのように書けばいいのでしょうか。たとえば、ドット (.) というメタ文字とマッチする正規表現というのは、いったいどう書くのでしょう。

メタ文字は、自分自身という意味を持っていませんので、メタ文字自身をあらわす正規表現を書くためには、メタ文字が持っている本来の意味を、その文字自身という意味に変更しないといけません。文字が持っている本来の意味を別の意味に変更するすることを、文字を「エスケープする」(escape) と言います。

文字をエスケープしたいときは、バックスラッシュ(\ ) というメタ文字を使います。バックスラッシュというのは、その直後に書かれた文字をエスケープするという意味を持つ文字です。

バックスラッシュでメタ文字をエスケープすると、その文字の意味は、メタ文字としての意味から、その文字自身という意味に変更されます。たとえば、ドット (.) というのはメタ文字ですので、ドット自身ではなくてメタ文字としての意味を持っています。しかし、 \. というように、バックスラッシュの右側にドットを書くと、その 2 文字は、ドット自身をあらわす正規表現になります。

ちなみに、バックスラッシュ自身もメタ文字ですから、バックスラッシュ自身をあらわす正規表現は、 \\ と書く必要があります。

また、スラッシュ(/) という文字はメタ文字ではありませんが、正規表現リテラルを作るための特別な文字ですので、正規表現リテラルの中でスラッシュ自身を記述するためには、 \/ というように、バックスラッシュを使ってエスケープしないといけません。

バックスラッシュによってエスケープすることができるのは、メタ文字だけではありません。メタ文字ではない文字をエスケープするということも可能です。つまり、自分自身という本来の意味を持っている文字に対して別の意味を与える、ということもできるわけです。

メタ文字ではない文字のいくつかは、それをバックスラッシュでエスケープすると、その文字自身とは異なる意味を持つ正規表現になります。たとえば、英小文字の n はメタ文字ではありませんので、自分自身というのが本来の意味ですが、バックスラッシュでエスケープした \n という 2 文字は、改行 (newline) という文字をあらわす正規表現になります。同じように、 \r はキャリッジリターン (carriage return)、 \t はタブ (tab)、 \f は改ページ (formfeed) という文字をあらわす正規表現です。ちなみに、空白、タブ、改行、キャリッジリターン、改ページ、という 5

種類の文字は、総称して「ホワイトスペース」(white space) と呼ばれます。

4.6.7　文字クラス

文字の集合のことを「文字クラス」(character class) と呼びます。正規表現は、指定された文字クラスに属する任意の文字というパターンを表現することも可能です。

すべての文字の集合（ただし改行は除きます）という文字クラスに属する任意の文字というパターンは、ドット (.) というメタ文字によってあらわされます。たとえば、yu.iko という正規表現は、yu と iko とのあいだに 1 個の任意の文字がある、というパターンをあらわしています。

それでは、irb を使って確かめてみましょう。

```
/yu.iko/ =~ "yumiko"
```

という式を入力してみてください。yu.iko という正規表現は、yumiko という文字列の全体とマッチしますので、文字列の先頭を意味する 0 という整数が戻り値として返ってきます。右辺の文字列を、yukiko、yuriko、yu8iko、yu@iko などに変えても、同じように 0 が返ってくるはずです。しかし、ドットという正規表現は、あくまで 1 個の文字としかマッチしませんので、右辺が yushiko という文字列だと、戻り値として nil が返ってきます。

文字を列挙することによって文字クラスを指定したいときは、角括弧 ([]) というメタ文字を使います。文字クラスに属する文字を角括弧の中に列挙したものは、その文字クラスに含まれる任意の文字というパターンをあらわす正規表現になります。たとえば、yu[mkr]iko という正規表現は、yu と iko とのあいだに、m、k、r のうちのいずれかひとつがある、というパターンをあらわしています。ですから、

```
/yu[mkr]iko/ =~ "yumiko"
```

という式の値は 0 ですが、

```
/yu[mkr]iko/ =~ "yubiko"
```

という式の値は nil になります。

文字を列挙することによって文字クラスを指定するのではなくて、文字コードの範囲を指定することによって文字クラスを指定する、ということも可能です。

文字コードの範囲で文字クラスを指定したいときは、マイナス (-) というメタ文字を使います。角括弧の中に、

[文字 $_1$] - [文字 $_2$]

という形のものを書くと、それは、文字 $_1$ から文字 $_2$ までという文字コードの範囲に含まれるすべての文字を列挙したのと同じ意味になります。たとえば、[a-z] という正規表現は、任意の英字の小文字というパターンをあらわします。ですから、

```
/yu[a-z]iko/ =~ "yumiko"
```

という式の値は 0 になります。右辺の文字列を yukiko や yubiko や yuxiko などに変えても同じです。しかし、yuMiko や yu8iko や yu@iko などだと nil になります。

文字クラスを指定する方法には、それに属する文字について記述するという方法のほかに、それに属さない文字について記述するという方法もあります。

属さない文字を記述することによって文字クラスを指定したいときは、サーカムフレックス (^) というメタ文字を使います。左角括弧の直後にサーカムフレックスを書くと、文字クラスは、それに属する文字によって記述されるのではなくて、それに属さない文字によって記述されることになります。たとえば、[^d] という正規表現は、d という文字を除いたすべての文字、というパターンをあらわします。ですから、

```
/yu[^m]iko/ =~ "yumiko"
```

という式の値は nil になりますが、右辺の文字列を yukiko や yuxiko や yu8iko や yu@iko などに変えると、0 になります。

サーカムフレックスとマイナスとを組み合わせることも可能です。たとえば、[^A-Za-z] という正規表現は、英字以外のすべての文字、というパターンをあらわします。ですから、

```
/yu[^A-Za-z]iko/ =~ "yumiko"
```

という式の値は nil になります。右辺の文字列を yukiko や yuxiko や yuMiko などに変えても同じです。しかし、yu8iko や yu@iko などに変えると、0 になります。

略記法	もとの正規表現	説明
\d	[0-9]	数字
\D	[^0-9]	数字以外
\w	[0-9A-Za-z]	英数字
\W	[^0-9A-Za-z]	英数字以外
\s	[\t\n\r\f]	ホワイトスペース
\S	[^ \t\n\r\f]	ホワイトスペース以外

表 4.8: 文字クラスの略記法

文字クラスをあらわす角括弧の中では、大多数のメタ文字は、バックスラッシュでエスケープしなくても自分自身をあらわします。角括弧の中でもかならずエスケープしないといけないメタ文字は、バックスラッシュのみです。右角括弧 (]) は、左角括弧 ([) の直後に書いた場合、自分自身をあらわします。マイナス (-) は、左角括弧の直後または右角括弧の直前に書けば自分自身という意味になります。そして、サーカムフレックス (^) も、左角括弧の直後でなければ自分自身をあらわします。

なお、文字クラスのうちで、しばしば使われるいくつかのものについては、図 4.8 に示したような略記法があります。

4.6.8 繰り返し

メタ文字を使うことによって、同じパターンがいくつも繰り返されている、というパターンをあらわす正規表現を作ることも可能です。

0 回以上の繰り返しを表現したいときは、アスタリスク (*) というメタ文字を使います。何らかの正規表現の直後にアスタリスクを書いたものは、その正規表現によってあらわされたパターンが 0 回以上繰り返されたもの、というパターンを意味する正規表現になります。たとえば、m* という正規表現は、長さが 0 以上の m の列、というパターンをあらわします。ですから、

```
/yum*iko/ =~ "yuiko"
```

という式の値は 0 になります。右辺の文字列を yumiko や yummiko や yummmiko や yummmmiko などに変えても同じです。

同じように、[mk]* という正規表現は、m または k から構成される、長さが 0 以上の文字列とマッチします。それでは、

```
/yu[mk]*iko/ =~ "yuiko"
```

という式の右辺を、yumiko、yukiko、yukmiko、yumkiko、yukmmiko、yumkmmkiko などに変えて試してみてください。

1 回以上の繰り返しとはマッチするけれども、繰り返しの回数が 0 回のときはマッチしないようにしたい、というときは、アスタリスクの代わりにプラス (+) というメタ文字を使います。たとえば、m+ という正規表現は、長さが 1 以上の m の列、というパターンをあらわします。ですから、

```
/yum+iko/ =~ "yumiko"
```

という式の値は 0 で、右辺の文字列を yummiko や yummmiko や yummmmiko などに変えても同じですが、yuiko に変えると nil になります。

任意の回数の繰り返しではなくて、0 回と 1 回の繰り返しだけとマッチする正規表現を書きたい、というときは、クエスチョンマーク (?) というメタ文字を使います。たとえば、m? という正規表現は、空文字列または m、というパターンをあらわします。ですから、

```
/yum?iko/ =~ "yumiko"
```

という式の値は 0 で、右辺の文字列を yuiko に変えても同じですが、yummiko に変えると nil になります。

繰り返しの回数を整数で指定したいときは、中括弧 ({}) というメタ文字を使います。たとえば、a{4} という正規表現は、aaaa というパターンをあらわします。

n 回から m 回までの繰り返しをあらわす正規表現は、やはり中括弧を使って、{n,m} と書き

アンカー	説明
^	行の先頭（文字列の先頭または改行の直後）
$	行の末尾（文字列の末尾または改行の直前）
\A	^と同じ
\Z	$と同じ
\z	文字列の末尾（改行は意識しない）
\b	単語の境界（角括弧の中ではバックスペース）
\B	単語の境界ではない位置
\G	前回の探索でマッチした部分文字列の末尾
(?=r)	正規表現 r とマッチした位置
(?!r)	正規表現 r とはマッチしない位置

表 4.9: アンカー

ます。たとえば、a{4,7} という正規表現は、長さが 4 から 7 までの a の列、というパターンをあらわします。

n 回以上の任意の回数の繰り返しをあらわす正規表現は、{n,} と書きます。たとえば、a{4,} という正規表現は、長さが 4 以上の a の列、というパターンをあらわします。

繰り返しの対象となる正規表現の範囲は、繰り返しをあらわす記述の直前にあるものだけ、という点に注意してください。つまり、mi+ という正規表現で繰り返しの対象になるのは、mi ではなくて、i だということです。

いくつかの正規表現の列を繰り返しの対象として指定したいときは、丸括弧というメタ文字で、指定したい正規表現の列を囲みます。たとえば、(mi)+ という正規表現は、mi という文字列を 1 回以上繰り返したもの、というパターンをあらわします。ですから、

```
/yu(mi)+ko/ =~ "yumiko"
```

という式の値は 0 で、右辺の文字列を yumimiko や yumimimiko などに変えても同じです。

4.6.9　選択

二つのパターンのうちのどちらか、というパターンの選択を表現したいときは、縦棒 (|) というメタ文字を使います。

縦棒を使って選択を記述したいときは、

正規表現の列 | 正規表現の列

という形の正規表現を書きます。そうすると、縦棒の左右に書かれたそれぞれの正規表現の列があらわしているパターンのどちらか、という意味の正規表現になります。たとえば、a|b という正規表現は、a または b のどちらか、というパターンをあらわします。選択の対象がさらに選択であってもかまいませんので、a|b|c という正規表現を書くことによって、a または b または c のいずれか、というパターンをあらわすことも可能です。

縦棒による選択の範囲は、その直前と直後の正規表現だけではなくて、そのさらに左や右にも及ぶ、という点に注意してください。つまり、ab|cd という正規表現とマッチする文字列は、abd と acd ではなくて、ab と cd だということです。

選択の対象となる正規表現の範囲を狭く限定したいときは、その範囲を丸括弧で囲みます。たとえば、a(b|c)d という正規表現は、abd と acd という二つの文字列とマッチします。同じように、ma(ri|sa)ko という正規表現は、mariko と masako という二つの文字列とマッチしますので、

```
/ma(ri|sa)ko/ =~ "mariko"
```

という式の値は 0 で、右辺の文字列を masako に変えても、やはり 0 です。

4.6.10 アンカー

パターンが文字列の中の特定の位置にあるときだけマッチする正規表現を書きたいときは、文字列の中の特定の位置を指定する正規表現を使います。そのような正規表現は、「アンカー」(anchor) と呼ばれます。表 4.9 は、アンカーにはどのようなものがあるかということを示したものです。

たとえば、行の先頭（文字列の先頭または改行の直後）という位置を指定したいときは、サーカムフレックス (^) というメタ文字をアンカーとして使います。たとえば、

```
/xyz/ =~ "cccxyzccc\nxyzccc"
```

という式の値は 3 ですが、

```
/^xyz/ =~ "cccxyzccc\nxyzccc"
```

という式の値は 10 になります。

行の末尾（文字列の末尾または改行の直前）という位置を指定したいときは、ドルマーク ($) というメタ文字をアンカーとして使います。たとえば、

```
/xyz/ =~ "cccxyzcccxyz\nccc"
```

という式の値は 3 ですが、

```
/xyz$/ =~ "cccxyzcccxyz\nccc"
```

という式の値は 9 になります。

4.6.11 部分文字列の置き換え

文字列のオブジェクトは、正規表現を扱うことのできるさまざまなメソッドを持っています。それらのメソッドを使うことによって、単にパターンを探し出すだけではなくて、発見されたパターンに対してさまざまな処理を実行することができます。

第 1.4 節と第 2.9 節で紹介した gsub も、正規表現を扱うことのできるメソッドのひとつです。

gsub は、レシーバーの中で部分文字列を探索して、発見された部分文字列を別の文字列に置き換えることによってできる文字列を戻り値として返すメソッドです。実は、gsub が探索する部分文字列というのは、正規表現で指定することも可能なのです。

1 個目の引数として正規表現を渡して gsub を呼び出すと、gsub は、その正規表現とマッチしたすべての部分文字列を、2 個目の引数またはブロックの中の式の値に置き換えます。たとえば、

```
"ccc3017cc529cccc84122cc".gsub(/[0-9]+/, "///")
```

という式を評価したとすると、その値として、

```
"ccc///cc///cccc///cc"
```

という文字列が得られます。

次のプログラムは、読み込んだ文字列のうちで、指定されたパターンとマッチする部分文字列を別の文字列に置き換えたものを標準出力に書き込む、という動作をするフィルターです。

プログラムの例 replace.rb

```
re = Regexp.new(ARGV.shift)
s = ARGV.shift
ARGF.read.gsub(re, s).display
```

実行例

```
$ ruby replace.rb [a-z] x replace.rb
xx = Rxxxxx.xxx(ARGV.xxxxx)
x = ARGV.xxxxx
ARGF.xxxx.xxxx(xx, x).xxxxxxx
```

4.6.12 文字列の分解

文字列のオブジェクトは、引数として与えられた正規表現にしたがってレシーバーをいくつかの部分に分解して、それらの部分から構成される配列を戻り値として返すメソッドを、二つ持っています。ひとつは scan、もうひとつは split です。

scan は、引数として正規表現を受け取って、レシーバーから、引数とマッチしたすべての部分文字列を取り出して、それらから構成される配列を戻り値として返すメソッドです。たとえば、

```
"ccc4372ccccc691cccc80115ccc".scan(/[0-9]+/)
```

という式を評価すると、

```
["4372", "691", "80115"]
```

という配列が値として得られます。

split は、第 4.2 節で紹介したように、レシーバーをいくつかの部分に区切って、それぞれの部分から構成される配列を戻り値として返すメソッドです。

引数として正規表現を渡して split を呼び出すと、split は、それがあらわしているパターンでレシーバーを区切ります。たとえば、

```
"east, west ,  north  ,south".split(/ *, */)
```

という式を評価すると、

```
["east", "west", "north", "south"]
```

という配列が値として得られます。

ちなみに、/ *, */ という正規表現を引数として渡すのではなくて、

```
"east, west ,  north  ,south".split(",")
```

というように、"," という文字列を引数として渡して split を呼び出した場合は、

```
["east", " west ", " north ", "south"]
```

というように、コンマの前後の空白が、単語の前後に残されることになります。

第 5 章　ネットワーク

5.1　ネットワークの基礎

5.1.1　サーバーとクライアント

ネットワークを利用して通信をするプログラムのペアは、普通、両者の関係が対等ではなくて、一方はサービスを提供する側で、他方はそのサービスを利用する側、というように役割が分化しています。サービスを提供するプログラムは「サーバー」(server) と呼ばれ、サービスを利用するプログラムは「クライアント」(client) と呼ばれます。

サーバーは、コンピュータの上で常に動作し続けていて、クライアントからの接続を待っています。そして、クライアントが自分に接続して何らかの要求を送ってきた場合、その要求にしたがってクライアントにサービスを提供します。クライアントがサーバーに接続してから、その接続を終了するまでの期間に両者のあいだで交される一連の通信は、「セッション」(session) と呼ばれます。

プログラムとプログラムとが通信をするためには、両者が、あらかじめ定められている通信のための規約にしたがって動作する必要があります。そのような規約は、「プロトコル」(protocol) と呼ばれます。標準として使われるプロトコルについては、IETF(Internet Engineering Task Force) という名前の組織で議論されています[1]。

IETF は、標準として提案されたプロトコルを RFC(Request For Comments) と呼ばれる文書にして公開しています。それぞれの RFC には番号が与えられていて、RFC2822 とか RFC2396 というような記述で、特定の RFC に言及することができるようになっています[2]。

5.1.2　IP アドレス

ネットワークに接続されているコンピュータのことを「ホスト」(host) と呼びます。プログラムとプログラムとが通信をするためには、相手のプログラムが動作しているホストを識別する必要があります。

ホストはかならず、ほかのホストが自分を識別することができるように、「IP アドレス」(IP address) と呼ばれる番号を持っています（RFC791）。

---

[1] IETF の URL は http://www.ietf.org/ です。

[2] RFC は、http://www.rfc-editor.org/rfc/ などのサイトからダウンロードすることができます。

IP アドレスは、32 個のビットから構成される列です。それを書きあらわすときは、普通、それを 8 ビットずつに 4 等分して、それぞれの部分を 10 進数にして、それらをドット (.) で区切ります。たとえば、

```
11010011000001100110110100110010
```

という IP アドレスは、

```
211.6.109.50
```

と書きあらわされます。

5.1.3 ホスト名

IP アドレスというのは人間にとっては扱いにくいものなので、英字や数字などを使って作られた「ホスト名」(host name) と呼ばれる名前を使ってホストを識別することもできるようになっています。

しかし、膨大な台数のホストが接続されているネットワークでは、それぞれのホストのホスト名を管理することがきわめて困難です。そのため、大規模なネットワークでは、ネットワークを「ドメイン」(domain) と呼ばれる領域に分割して、それぞれのドメインごとにホスト名を管理することができるようにしています。ドメインは、その中をさらにドメインに分割することも可能です。ですから、ドメインというのは、ディレクトリと同じように階層的な構造を持つことになります。

ドメインは、「ドメイン名」(domain name) と呼ばれる名前によって識別されます。そして、ネットワーク上のホストは、ドメイン名によって修飾されたホスト名によって識別されることになります。

ドメイン名でホスト名を修飾したいときは、左端にホスト名を書いて、その右側に、階層の下にあるものから上にあるものへ順番にドメイン名を並べていって、それらのあいだをドット (.) で区切ります。つまり、

ホスト名 . ドメイン名 . ･･･ . ドメイン名

というように書くわけです。たとえば、`com` というドメインの中の `example` というドメインの中の `www` というホストを指定したいときは、

```
www.example.com
```

という名前を書けばいいわけです。

なお、個々のホストに付けられた名前というのが「ホスト名」という言葉の本来の意味ですが、普通、ホスト名とドメイン名から構成される名前の列のことも「ホスト名」と呼ばれます。

5.1.4 ポート

クライアントがサーバーに接続するためには、そのサーバーが動作しているホストを指定する必要があるわけですが、それだけではまだ充分ではありません。ひとつのホストの上で動作しているサーバーはひとつだけとは限らないからです。

ホストが持っている通信のための出入口のことを「ポート」(port) と呼びます。1 台のホストはいくつものポートを持っていて、サーバーは、それらのポートのうちのどれかひとつを使ってクライアントからの接続を待っています。ですから、クライアントは、ポートを指定することによって特定のサーバーに接続することができるわけです。

ポートは、「ポート番号」(port number) と呼ばれる番号によって識別されます。ポート番号は、0 から 65535 までの整数です。標準として使われるプロトコルは、何番のポートを使うのかということも定めていて、それらのポートは「well-known ポート」(well-known port) と呼ばれます。

5.1.5 ソケット

プログラムにとって、通信の相手となるプログラムは、そこからデータを読み込んだりそこへデータを書き込んだりすることのできる対象として見えます。プログラムが別のプログラムと通信をするためには、通信の相手をあらわすオブジェクトを作る必要があります。そのようなオブジェクトは、「ソケット」(socket) と呼ばれます。

Ruby では、データの読み書きの対象は、`IO` というクラスのインスタンスによってあらわされます。`IO` クラスのインスタンスは「IO オブジェクト」と呼ばれ、それは、データを読み込んだ

り書き込んだりするためのさまざまなメソッドを持っています。

ソケットというのもデータの読み書きの対象ですので、Ruby では、各種のソケットのクラスは、すべて IO クラスを継承しています。つまり、Ruby のソケットは IO オブジェクトの一種だということです。したがって、相手のプログラムにデータを送信したり、送られてきたデータを受信したりしたいときは、ソケットのクラスが IO クラスから継承した読み書きのメソッドを、どれでも使うことができます。

各種のソケットのクラスは Ruby の組み込みクラスではありませんので、それらのクラスを利用するプログラムは、それらを定義しているライブラリーを読み込む必要があります。各種のソケットのクラスは、socket という名前のライブラリーの中で定義されています。ですから、プログラムの先頭に、

```
require("socket")
```

という式を書いておけば、各種のソケットのクラスを利用することができるようになります。

5.1.6 TCP

ネットワークを使って通信をするプログラムは、地層のように積み重なったいくつかのプロトコルを使うことになります。ネットワークで利用されるさまざまなサービスは、それぞれのサービスごとのプロトコルを持っているわけですが、それらのプロトコルのひとつ下の層としては、多くの場合、TCP(Transmission Control Protocol) という名前のプロトコル（RFC793）が使われています。

TCP を使って通信をするクライアントを書きたいときは、TCPSocket というクラスから生成されたソケットを使います。このクラスは、open というクラスメソッドを持っていて、このメソッドを使うことによって、クライアントから見た TCP による通信の相手をあらわすソケットを生成することができます。

TCPSocket クラスの open は、h が通信先のホスト名で、p が利用したいサービスのポート番号だとするとき、

```
TCPSocket.open(h, p)
```

という式で呼び出します。そうすると、open は、指定されたホストの指定されたポートでサービスを提供しているプログラムをあらわすソケットを生成して、それを戻り値として返します。たとえば、

```
sock = TCPSocket.open("www.example.com", 80)
```

という式を評価することによって、www.example.com というホストの 80 番のポートでサービスを提供しているプログラムをあらわすソケットを生成して、それを sock という変数に代入することができます。

ソケットを生成したあとは、それが持っている読み込みのメソッドを使ってデータを受信したり、書き込みのメソッドを使ってデータを送信したりすることができます。

そして、通信が終了したときは、かならずその後始末をしないといけません。ファイルの後始末と同じように、ソケットの後始末をすることも、ソケットを「クローズする」(close) と言います。ソケットのクローズは、ソケットが持っている close というメソッドを使うことによって実行することができます。たとえば、sock という変数がソケットを指し示しているとするならば、

```
sock.close
```

という式を評価することによって、ソケットをクローズすることができます。

5.1.7 改行

どのような文字を使って改行をあらわすかというのは、オペレーティングシステムごとに異なっています。MacOS では文字コードが 13 の CR(carriage return) と呼ばれる文字が使われ、UNIX では文字コードが 10 の LF(line feed) と呼ばれる文字が使われ、Windows では、CRLF、つまり CR と LF から構成される文字列が使われます。

しかし、プログラムが別のプログラムと通信をするときに、相手のオペレーティングシステムによって改行の文字を使い分ける必要はありません。なぜなら、通信で使われる改行は CRLF に統一されているからです。

文字列をあらわすリテラルの中では、CR は \r という記述であらわされ、LF は \n という記

述であらわされます。ですから、"\r\n" というリテラルを書くことによって、CRLF をあらわすことができます。

受信した行の末尾に付いている CRLF を取り除きたいときは、文字列が持っている chomp というメソッドに、引数として "\r\n" を渡します。たとえば、sock という変数がソケットを指し示しているとするとき、

```
line = sock.gets.chomp("\r\n")
```

という式を書くことによって、1 個の行を受信して、その行末から CRLF を取り除いたものを line に代入する、ということができます。

5.1.8 クライアントのクラス

次のプログラムは、クライアントをあらわす Client というクラスを定義します。

プログラムの例 client.rb

```
require("socket")

class Client
  def initialize(host, port)
    @socket = TCPSocket.open(host, port)
  end
  def close
    @socket.close
  end
  def read(length)
    if length == 0
      @socket.read
    else
      @socket.read(length)
    end
  end
  def send(s)
    ("C: " + s + "\n").display
    @socket.write(s + "\r\n")
  end
  def receive
    line = @socket.gets.chomp("\r\n")
    ("S: " + line + "\n").display
    line
  end
  def sendReceive(s)
    send(s)
    receive
  end
end
```

このクラスの中で定義されている read というメソッドは、引数としてデータの大きさを受け取って（単位はバイト）、その大きさのデータをサーバーから受信します。引数として 0 を渡すと、セッションが終了するまでデータを受信し続けます。

send というメソッドは、引数として文字列を受け取って、それを標準出力に出力して、同じものをサーバーへ送信します。

receive というメソッドは、サーバーから 1 行の文字列を受信して、それを標準出力に出力して、同じものを戻り値として返します。

5.2 HTTP

5.2.1 HTTP の概要

「ウェブ」(web) あるいは「WWW」(world wide web) と呼ばれるサービスを取り扱うプログラムは、HTTP(Hypertext Transfer Protocol) と呼ばれるプロトコル（RFC2616）を使って通信をします。なお、HTTP にはいくつかのバージョンがあって、それぞれのバージョンの HTTP は、HTTP/1.0 とか HTTP/1.1 というように、バージョン番号を含んだ名前で呼ばれます。

特定のプロトコルを使って通信をするサーバーとクライアントのそれぞれは、普通、そのプロトコルの名前を頭に付けて、「何々サーバー」とか「何々クライアント」と呼ばれます。ですから、HTTP を使って通信をするサーバーとクライアントは、「HTTP サーバー」と「HTTP クライアント」と呼ばれることになります。

HTTP サーバーは、通常、80 番のポートを使って HTTP クライアントと通信をします。

HTTP サーバーが提供しているサービスというのは、基本的には、ファイルの内容をクライアントに送信することです。HTTP サーバーは、ファイルの内容の要求を意味するデータを HTTP クライアントから受け取った場合、要求されたファイルの内容をクライアントに送信します。

HTTP クライアントが何らかのサービスを要求するためにサーバーに送るデータは、「リクエスト」(request) と呼ばれます。そして、リクエストに応えて HTTP サーバーがクライアントに送るデータは、「レスポンス」(response) と呼ばれます。

5.2.2 リクエスト

HTTP クライアントが送信するリクエストは、「リクエスト行」(request line) と呼ばれるひとつの行で始まります。リクエスト行は、

メソッド名 パス名 バージョン CRLF

という形式の文字列です。「メソッド名」のところには、サーバーが持っている機能を識別する名前を指定します。たとえば、ファイルの内容を要求する場合は、GET というメソッド名を指定します。

「パス名」のところには、ファイルの絶対パス名を指定します。GET メソッドを指定した場合は、ここで指定したファイルの内容が送られてくることになります。

サーバーの設定によっては、パス名のところに、ファイルではなくてディレクトリのパス名を指定することもできます（ただし、末尾にスラッシュを付ける必要があります）。それが可能になっている場合は、指定されたディレクトリにある、サーバーで設定されている名前（たとえば index.html など）のファイルの内容が送られてきます。

「バージョン」のところには、バージョン番号を含んだ HTTP の名前、つまり、HTTP/1.0 とか HTTP/1.1 というような文字列を指定します。

ですから、たとえば、HTTP/1.1 を使ってルートディレクトリの下にある namako.htm というファイルの内容を要求したいとすると、クライアントは、

```
GET /namako.htm HTTP/1.1
```

というリクエスト行を送信すればいいわけです。

HTTP クライアントは、リクエスト行に続けて、「ヘッダー」(header) と呼ばれる何行かの文字列を送信します。ヘッダーを構成するそれぞれの行は、「フィールド」(field) と呼ばれます。フィールドは、

フィールド名 : フィールド本体 CRLF

という形式の文字列です。たとえば、User-Agent という名前のフィールドを使うことによって、

```
User-Agent: Gamera/6.0
```

というように、クライアントの名前やバージョンを送信することができます。

HTTP/1.1 では、Host というフィールドはかならず送信しないといけない、と定められています。それ以外のフィールドは、送信してもしなくてもどちらでもかまいません。Host フィールドというのは、サーバーが動作しているホストのホスト名を指定するフィールドで、たとえば、

```
Host: www.example.com
```

というようなものを送信します。

HTTP クライアントは、ヘッダーの送信が終了したのち、そのことをサーバーに通知するために、空行（改行だけの行）を送信する必要があります。

HTTP サーバーに対してファイルの内容を要求する場合は、以上の三つのもの、つまり、リクエスト行、ヘッダー、そして空行をリクエストとして送信します。そうすると、その結果として、ファイルの内容を含むレスポンスがサーバーから送られてきます。

5.2.3 レスポンス

HTTP サーバーが送信するレスポンスの構造は、リクエストの構造とよく似ています。

レスポンスは、「ステータス行」(status line) と呼ばれる行で始まります。サーバーは、その次にヘッダーを送信して、ヘッダーが終わったのち、空行を送信します。そして、GET メソッドのリクエストに対応するレスポンスの場合は、その空行に続けて、ファイルの内容を送信します。

ステータス行は、

バージョン	ステータスコード	理由	CRLF

という形式の文字列です。「バージョン」のところは、HTTP/1.1 というような、バージョン番号を含んだ HTTP の名前です。

「ステータスコード」(status code) というのは、リクエストがどのように処理されたかということをあらわす 3 桁の 10 進数のことです。たとえば、処理が成功した場合のステータスコードは 200、ファイルが見付からないというエラーが発生した場合のステータスコードは 404、というように、さまざまなステータスコードが定められています。

「理由」のところは、ステータスコードがあらわしているものについての簡潔な説明です。たとえば、200 の場合は OK、404 の場合は Not Found というような文字列になります。ですから、HTTP/1.1 のサーバーは、要求された処理に成功した場合、

```
HTTP/1.1 200 OK
```

というステータス行を送信することになります。

HTTP サーバーは、ステータス行に続けて、何行かのフィールドから構成されるヘッダーを送信します。それらのフィールドのうちでもっとも重要なのは、送信するデータの大きさ（単位はバイト）を通知する Content-Length というフィールドです。たとえば、

```
Content-Length: 3876
```

というフィールドは、空行のあとに送信するデータの大きさが 3876 バイトだということを示しています。

5.2.4 HTTP クライアントの例

それでは、Ruby を使って HTTP クライアントを書いてみましょう。

プログラムの例 http.rb

```
require("client")

class HTTP < Client
  def initialize(host, path)
    super(host, 80)
    @host = host
    @path = path
  end
  def sendRequest
    send("GET " + @path + " HTTP/1.1")
    send("Host: " + @host)
    send("")
  end
  def receiveHeader
    length = 0
    while (field = receive) != ""
      if /^Content\-Length/ =~ field
        length = field.scan(/[0-9]+/)[0].to_i
      end
    end
    length
  end
  def receiveResource(length)
    if @path[@path.size - 1, 1] == "/"
      filename = "index.html"
    else
      filename = File.basename(@path)
    end
    open(filename, "w") do |outf|
```

```
      outf.write(read(length))
    end
  end
  def receiveResponse
    receiveResource(receiveHeader)
  end
end

if ARGV.size == 2
  http = HTTP.new(ARGV[0], ARGV[1])
  http.sendRequest
  http.receiveResponse
  http.close
else
  "使い方: ruby http.rb ホスト名 パス名\n".display
end
```

このプログラムを、たとえば、

```
ruby http.rb www.example.com /namako.htm
```

というコマンドで実行したとすると、www.example.comというホストのHTTPサーバーから、/namako.htmというファイルの内容が送られてくることになります。ただし、このホストとファイルは実在しませんので、実在するホストとファイルを指定して試してみてください。

5.3 SMTP

5.3.1 MTAとMDAとMUA

ホストのユーザーが別のユーザーに対してデータを送ったり受け取ったりするためのシステムは、「電子メール」(electronic mail) または単に「メール」(mail) と呼ばれます。そして、メールを使ってユーザーとユーザーとのあいだで交換されるデータは、「メールメッセージ」(mail message) と呼ばれます。

メールというサービスを実現するためのプログラムは、MTA、MDA、MUAと呼ばれる3種類のプログラムに分類することができます。

MTA(mail transfer agent) というのは、ホストからホストへメールメッセージを転送するプログラムのことです。MTAは、受け取ったメールメッセージを別のホストのMTAに転送するか、またはMDAに渡します。

MDA(mail delivery agent) というのは、MTAからメールメッセージを受け取って、それをユーザーのメールボックスに配達するプログラムのことです。

MUA(mail user agent) というのは、ユーザーがメールメッセージを取り扱うためのプログラムのことです。ユーザーは、MUAを使うことによって、メールメッセージを書いて送信したり、メールメッセージを受け取って閲覧したりすることができます。

MTAが別のホストのMTAにメールメッセージを送信するときや、MUAがMTAにメールメッセージを送信するときには、SMTP(Simple Mail Transfer Protocol) というプロトコル（RFC2821）が使われます。そして、メールボックスが置かれているホストとは異なるホストでMUAを使っている場合には、メールボックスからメールメッセージをダウンロードするために、POP3(Post Office Protocol Version 3) というプロトコル（RFC1939）またはIMAP(Internet Message Access Protocol) というプロトコル（RFC2060）が使われます。

5.3.2 メールメッセージの書き方

次に、メールメッセージというものはどのように書けばいいのかということについて、ごく簡単に説明しておくことにしましょう。なお、メールメッセージの基本的な書き方については、RFC2822というRFCで規定されていますので、もっと詳しく知りたい方はそちらを参照してください。

メールメッセージの全体は、何個かの行から構成される文字列です。改行は、CRLFであらわされます。

メールメッセージは、先頭に「ヘッダー」(header) と呼ばれる部分があって、その下に「本文」(body) と呼ばれる部分が続きます。ヘッダーと本文とのあいだは、空行（改行だけの行）で区切ります。

```
To: yoshihiko@example.com
From: risako@example.com
Subject: Chinese restaurant
MIME-Version: 1.0
Content-Type: text/plain; charset=iso-2022-jp

この度、脱サラして中華料理の店をオープンしました。
ぜひ食べに来てください。
```

図 5.1: メールメッセージの例

ヘッダーというのは、メールメッセージ自身についての情報（送信者、受信者、主題など）を書くための部分です。ヘッダーは、「フィールド」(field) と呼ばれる文字列から構成されます。

フィールドにはさまざまな種類があって、それらの種類は、「フィールド名」(field name) と呼ばれる名前によって識別されます。たとえば、送信者のアドレスを書くフィールドのフィールド名は、`From` です。同じように、受信者は `To`、主題は `Subject`、というようにフィールド名が決まっています。

ひとつのフィールドは、

フィールド名 : フィールド値 CRLF

という形式で書きます。たとえば、送信者のアドレスのフィールドは、

```
From: sayaka@example.com
```

というように書くことになります。

ヘッダーは、ASCII と呼ばれる文字コードを使って書かないといけないことになっています。ASCII は仮名や漢字を含んでいませんので、ヘッダーの中で日本語を使うためには、MIME と呼ばれる規定（RFC2045–2049）にしたがって、日本語の文字を ASCII の文字に変換する必要があります。

メールメッセージの本文というのは、送信者が受信者に伝えたい内容を書くための部分です。この部分には、日本語の文章を書いてもかまいません。ただし、その場合に使うことのできる日本語の文字コードは、JIS のみです[3]。メールメッセージの本文にしたい文章が、Shift_JIS や EUC–JP などの JIS 以外の文字コードで書かれている場合は、送信する前にそれを JIS に変換する必要があります。

`kconv` という名前の Ruby のライブラリーは、日本語の文字コードを変換するためのいくつかのメソッドを定義しています。その中にある `tojis` というメソッドを使うと、Shift_JIS または EUC–JP を JIS に変換することができます。`tojis` は、文字列のオブジェクトの中に作られるメソッドで、レシーバーの文字コードを JIS に変換した結果を戻り値として返します。

なお、JIS を使ってメールメッセージの本文を書く場合は、ヘッダーの中に、

```
MIME-Version: 1.0
Content-Type: text/plain; charset=iso-2022-jp
```

という二つのフィールドを書いておく必要があります。

さて、メールメッセージの書き方についての説明は、以上で終わりです。図 5.1 に、メールメッセージの一例を書いておきましたので、参考にしてください。

5.3.3 SMTP のセッション

MTA は、SMTP(Simple Mail Transfer Protocol) というプロトコルを使って、メールメッセージを別のホストへ転送します。また、MUA も、MTA にメールメッセージを送信するときに同じプロトコルを使います。SMTP を使ってメールメッセージを転送する場合、受信するほうのプログラムがサーバーになって、送信するほうのプログラムがクライアントになります。SMTP サーバーは、通常、25 番のポートを使って SMTP クライアントと通信をします。

SMTP クライアントは、「コマンド」(command) と呼ばれる文字列を SMTP サーバーに送ることによって、サーバーに対してさまざまな情報を通知します。SMTP には、表 5.1 のようなコマンドがあります。

[3] JIS という文字コードの正式な名称は ISO–2022–JP です。

コマンド	説明
helo ホスト名	クライアントが動作しているホストの名前を通知します。
mail from:<アドレス>	送信者のアドレスを通知します。
rcpt to:<アドレス>	受信者のアドレスを通知します。
data	これからメールメッセージを送信するということを通知します。
quit	セッションを終了するということを通知します。

表 5.1: SMTP の主要なコマンド

サーバーは、受け取ったコマンドを処理したのち、「応答」(reply) と呼ばれる文字列をクライアントに送ります。

SMTP では、セッションを進行させる順序がかなり厳格に決まっていて、クライアントは、helo、mail、rcpt、data、quit、という順序でコマンドを送信する必要があります。

1 回のセッションでは、メールメッセージを 1 通だけしか送信することができません。しかし、1 回のセッションで 1 通のメールメッセージを 2 人以上の受信者に送信する、ということは可能です。その場合は、それぞれの受信者のアドレスを引数とする 2 個以上の rcpt コマンドを送信します。

メールメッセージは、data コマンドを送信して、それに対する応答がサーバーから返ってきたのちに送信します。メールメッセージの送信が終了したとき、クライアントは、メールメッセージがこれで終わりだということをサーバーに通知するために、ドット (.) のみの行を送信しないといけません。サーバーは、メールメッセージの受信が終了したときも、クライアントに応答を返します。

5.3.4 SMTP クライアントの例

次のプログラムは、Ruby で書かれた SMTP クライアントの一例です。

プログラムの例 smtp.rb

```
require("client")
require("kconv")

class String
  def getFieldValue
    value = split(/:/)
    if value.size >= 2
      value[1].delete(" ")
    else
      ""
    end
  end
end

class SMTP < Client
  def initialize(argv)
    host, @path = argv
    super(host, 25)
    @to = @from = ""
  end
  def readHeader
    open(@path) do |inf|
      header = true
      while (line = inf.gets.chomp) && header
        if line == ""
          header = false
        elsif /^To/ =~ line
          @to = line.getFieldValue
        elsif /^From/ =~ line
          @from = line.getFieldValue
        end
      end
```

```
    end
  end
  def sendMail
    receive
    sendReceive("helo localhost")
    sendReceive("mail from:<" + @from + ">")
    sendReceive("rcpt to:<" + @to + ">")
    sendReceive("data")
    open(@path) do |inf|
      inf.each do |line|
        send(line.chomp.tojis)
      end
    end
    sendReceive(".")
    sendReceive("quit")
  end
end

if ARGV.size == 2
  smtp = SMTP.new(ARGV)
  smtp.readHeader
  smtp.sendMail
  smtp.close
else
  ("使い方: ruby smtp.rb ホスト名 パス名\n").display
end
```

それでは、エディターを使ってメールメッセージをファイルに保存したのち、SMTP サーバーのホスト名と、メールメッセージのパス名を指定して、このプログラムを起動してみてください。そうすると、そのメールメッセージが SMTP サーバーへ送信されるはずです。

第 6 章　GUI

6.1　GUI の基礎

6.1.1　GUI とは何か

機械などが持っている、人間に何かを報告したり人間からの指示を受け付けたりする部分のことを、「ユーザーインターフェース」(user interface) と言います。

コンピュータのプログラムも、たいていのものは何らかのユーザーインターフェースを持っています。プログラムのユーザーインターフェースとしては、現在、CUI と GUI と呼ばれる 2 種類のものが主流です。

CUI(character user interface) というのは、文字列を媒介とするユーザーインターフェースのことです。CUI の場合、プログラムは文字列を出力することによって人間に何かを報告し、人間はキーボードから文字列を入力することによってプログラムに指示を与えます。

それに対して、GUI(graphical user interface) は、図形を媒介とするユーザーインターフェースです。GUI の場合、プログラムはモニターの画面に図形を表示することによって人間に何かを報告し、人間はマウスなどの装置を使ってその図形を操作することによってプログラムに指示を与えます。

6.1.2　Tk

Ruby の処理系は GUI を作る機能を内蔵していませんので、GUI を持つプログラムを Ruby で書くためには、そのためのライブラリーを使う必要があります。

Ruby で使うことのできる GUI のライブラリーにはさまざまなものがあるのですが、この文章では、Tk というライブラリーを使って GUI を作る方法について説明していくことにします。

Tk というのは、John Ousterhout さんという人が作った GUI のライブラリーで、Tk という名前は、toolkit という単語を縮めたものです。

Tk は、もともとは Tcl というプログラミング言語で使うために作られたものなのですが、現在では、Tcl だけではなくて Perl や Ruby でもそれを使うことができます。それぞれの言語と Tk とを組み合わせたものは、Tcl/Tk、Perl/Tk、Ruby/Tk というように呼ばれます。

Ruby/Tk は、GUI を作るためのさまざまなクラスやモジュールから構成されていて、それらのクラスやモジュールは、tk というファイル名のライブラリーの中で定義されています。ですから、Ruby のプログラムの先頭に、

```
require("tk")
```

という式を書いておけば、そのプログラムは、Ruby/Tk の機能を使って GUI を作ることができるようになります。

6.1.3 イベントループ

GUI を持つプログラムは、人間からの指示を受け取って、その指示に対応する処理を実行する、ということを何度も繰り返します。そのような繰り返しは、「イベントループ」(event loop) と呼ばれます。

Tk では、Tk というモジュールが持っている mainloop というメソッドがイベントループになっています。Tk を使って GUI を作るプログラムは、かならずこのメソッドを呼び出す必要があります。

それでは、mainloop を呼び出すだけのプログラムを書いて、それを実行してみましょう。

プログラムの例 empty.rb

```
require("tk")
Tk.mainloop
```

このプログラムを実行すると、空のウィンドウがモニターに表示されるはずです。

6.1.4 ウィジェット

GUI を作るためのは、ウィンドウの上にさまざまな部品を取り付ける必要があります。そのような、GUI を作るための部品は、「ウィジェット」(widget) と呼ばれます。ちなみに、widget という単語は、window gadget という言葉を縮めたものです。ウィジェットには、ラベル、ボタン、キャンバス、エントリー、リストボックスなどのさまざまな種類があります。

Ruby では、ウィジェットは、「ウィジェットクラス」(widget class) と呼ばれるクラスのインスタンスによって表示されます。たとえば、ラベルという種類のウィジェットは、TkLabel というクラスのインスタンスによって表示されます。なお、「ウィジェット」という言葉は、画面の上に表示される部品という意味だけではなくて、それを表示しているオブジェクトという意味でも使われます。

ウィジェットは、ウィジェットクラスが持っている new というクラスメソッドを呼び出すことによって生成することができます。たとえば、

```
label = TkLabel.new
```

という式を書くことによって、ラベルのオブジェクトを生成して、それを label という変数に代入することができます。

「ラベル」(label) というのは、文字列（テキスト）をその上に表示することを目的として使われるウィジェットです。ウィジェットの上に表示される文字列を設定したいときは、それが持っている text というメソッドを使います。text に引数として文字列を渡すと、その文字列がウィジェットの上に表示されます。たとえば、label という変数にラベルが格納されているとするとき、

```
label.text("namako")
```

というメッセージ式を評価すると、そのラベルの上に namako という文字列が表示されます。

ちなみに、ウィンドウというのも、ウィジェットの種類のひとつです。Tk には、ウィンドウを生成するクラスとして、TkRoot と TkToplevel という二つのものがあります。TkRoot のインスタンスは「ルートウィジェット」(root widget) と呼ばれ、TkToplevel のインスタンスは「トップレベル」(toplevel) と呼ばれます。mainloop を呼び出すと、ひとつのウィンドウが自動的に表示されるのですが、このウィンドウはルートウィジェットです。

6.1.5 ジオメトリーマネージャー

ウィンドウ以外のウィジェットは、単独では画面の上に表示されません。それを画面に表示するためには、それをウィンドウの上に取り付ける必要があります。

土台となるウィジェット（たとえばウィンドウ）の上にウィジェットを取り付けたいときは、「ジオメトリーマネージャー」(geometry manager) と呼ばれるものを使います。ジオメトリーマネージャーというのは、ウィジェットが持っているメソッドで、ウィジェットの配置を管理するという動作をするもののことです。

Tk には、pack、grid、place という 3 種類のジオメトリーマネージャーがあります。それらのうちで、もっともよく使われるのは、pack というジオメトリーマネージャーです。pack は、土台となるウィジェットを必要最小限に小さくして、その中にウィジェットを詰め込む、という動作をします。

プログラムの例 label.rb

```
require("tk")
label = TkLabel.new
label.text("私はラベルです。")
label.pack
Tk.mainloop
```

この pack というジオメトリーマネージャーについては、第 6.3 節で、もう少し詳しく説明したいと思います。

6.1.6 ウィジェットの初期設定

new を使ってウィジェットを生成するとき、ブロックを new に渡すと、そのブロックは、生成されたウィジェットの中の initialize によって実行されます。通常、ウィジェットに表示させる文字列を設定したり、ウィジェットを土台に取り付けたりする、というようなウィジェットの初期設定は、new に渡すブロックの中に書いておきます。なお、そのブロックの中では、メッセージ式のレシーバーを省略すると、メッセージは、新しく生成されたウィジェットに送られます。

次のプログラムは、先ほどのプログラムを、ブロックを使う形に書き換えたものです。

プログラムの例 label2.rb

```
require("tk")
TkLabel.new do
  text("私はラベルです。")
  pack
end
Tk.mainloop
```

Tk を使うプログラムでは、ルートウィジェットは暗黙のうちに生成されるわけですが、それを初期設定したいときは、それを生成する記述を明示的に書く必要があります。ルートウィジェットというのは TkRoot というクラスのインスタンスですので、

TkRoot.new ブロック

という式を書くことによって、ルートウィジェットを初期設定することができます。

ルートウィジェットのタイトルバーに表示されるタイトルを設定したいときは、title というメソッドを使います。title に引数として文字列を渡すと、その文字列がタイトルバーの上に表示されます。

プログラムの例 title.rb

```
require("tk")
TkRoot.new do
  title("I am a root widget.")
end
Tk.mainloop
```

6.1.7 フォント

Ruby/Tk では、ウィジェットの上に表示される文字のフォントは、TkFont というクラスのインスタンスとして取り扱われます。

TkFont クラスのインスタンスを生成する new は、フォントの属性を指定するハッシュを引数として受け取ります。そのハッシュは、表 6.1 に示したようなキーと値から構成されます。たとえば、

キー	値
size	大きさ（単位はポイント）
family	ファミリー（times、helvetica、courier など）
weight	太さ（normal または bold）
slant	傾き（roman または italic）

表 6.1: フォントの属性を指定するキーと値

```
TkFont.new("size"=>24, "weight"=>"bold", "slant"=>"italic")
```

という式を書くことによって、大きさ 24 ポイント、太字、イタリック、というフォントで文字を表示するための TkFont クラスのインスタンスを生成することができます。

ウィジェットにフォントを設定したいときは、ウィジェットが持っている font というメソッドを使います。font は、TkFont クラスのインスタンスを引数として受け取って、それをレシーバーに設定します。

プログラムの例 font.rb

```
require("tk")
times = TkFont.new("family"=>"times", "size"=>18)
helvetica = TkFont.new("family"=>"helvetica", "size"=>18)
TkLabel.new do
  text("このフォントは Times です。")
  font(times)
  pack
end
TkLabel.new do
  text("このフォントは Helvetica です。")
  font(helvetica)
  pack
end
Tk.mainloop
```

6.1.8 色

ウィジェットが持っているメソッドを使うことによって、ウィジェットの背景色や前景色（文字の色）を設定することができます。

Ruby/Tk では、色は、

#赤緑青

という形の文字列によって記述されます。赤、緑、青という三つの場所には、光の三原色のそれぞれの明るさを指定する 2 桁の 16 進数を書きます。たとえば、#ff0000 と書けば赤色、#00ffff と書けば水色、#ff8000 と書けばオレンジ色をあらわすことになります。

ウィジェットの背景色は、background というメソッドを呼び出すことによって設定することができて、ウィジェットの前景色は、foreground というメソッドを呼び出すことによって設定することができます。これらのメソッドは、色をあらわす文字列を引数として受け取って、その色をレシーバーに設定します。

プログラムの例 color.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
TkLabel.new do
  text("背景はネイビーで文字は黄色。")
  font(f)
  background("#000080")
  foreground("#ffff00")
  pack
end
Tk.mainloop
```

6.2 ボタン

6.2.1 普通のボタン

人間によってクリックされたときに何らかの動作を実行するウィジェットは、「ボタン」(button) と呼ばれます。

ボタンは、TkButton というクラスのインスタンスです。ボタンの上に表示する文字列は、ラベルの場合と同じように、text というメソッドを使うことによって設定することができます。

ボタンには、クリックされたときに実行する動作を設定する必要があります。動作の設定には、ボタンが持っている command というメソッドを使います。command にブロックを渡すと、command は、レシーバーに対して、クリックされたときにそのブロックを実行するという設定をします。たとえば、

```
command { a = 3 }
```

という式で動作を設定したとすると、ボタンをクリックしたときに、a という変数に 3 が代入されることになります。

プログラムの例 countup.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
count = 0
countlabel = TkLabel.new do
  text("0")
  font(f)
  pack
end
TkButton.new do
  text("増やす")
  font(f)
  command do
    count += 1
    countlabel.text(count.to_s)
  end
  pack
end
Tk.mainloop
```

6.2.2 Tk 変数オブジェクト

ボタンには、「チェックボタン」(check button) と「ラジオボタン」(radio button) と呼ばれる 2 種類の変種があります[1]。普通のボタンとそれらの変種との最大の相違点は、普通のボタンは状態を持たないのに対して変種のほうは状態を持つ、というところにあります。

チェックボタンやラジオボタンを使うためには、あらかじめ、その状態を保持するための箱を作っておく必要があります。ウィジェットの状態を保持するための箱というのは、「Tk 変数オブジェクト」(Tk variable object) と呼ばれるオブジェクトです。このオブジェクトは、ウィジェットの状態として 1 個の文字列を保持することができるようになっています。

Tk 変数オブジェクトは、TkVariable というクラスのインスタンスです。TkVariable クラスの new は、引数として 1 個の文字列を受け取って、Tk 変数オブジェクトを生成して、それが保持する状態の初期値として引数を設定します。たとえば、

```
var = TkVariable.new("namako")
```

という式を評価すると、namako という文字列が初期値として設定された Tk 変数オブジェクトが生成されて、それが var という変数に代入されます。

Tk 変数オブジェクトから、それが状態として保持している文字列を取り出したいときは、Tk 変数オブジェクトが持っている value というメソッドを使います。value は、レシーバーが状態として保持している文字列を戻り値として返します。

Tk 変数オブジェクトが状態として保持している文字列を変更したいときは、Tk 変数オブジェクトが持っている value= というメソッドを使います。value は、引数として文字列を受け取っ

[1] チェックボタンは、「チェックボックス」(check box) と呼ばれることもあります。

て、それをレシーバーに設定します。たとえば、 var という変数に Tk 変数オブジェクトが代入されているとするとき、

```
var.value = "umiushi"
```

という式を評価すると、その Tk 変数オブジェクトが保持している文字列が umiushi に置き換わります。

Tk 変数オブジェクトをラベルに設定すると、そのラベルは、設定された Tk 変数オブジェクトの内容を表示するようになります。Tk 変数オブジェクトをラベルに設定したいときは、ラベルが持っている textvariable というメソッドを使います。 textvariable は、引数として Tk 変数オブジェクトを受け取って、それをレシーバーに設定します。

Tk 変数オブジェクトをチェックボタンやラジオボタンに設定したいときは、それらのボタンが持っている variable というメソッドを使います。 variable は、Tk 変数オブジェクトを引数として受け取って、それをレシーバーに設定します。

6.2.3 チェックボタン

チェックボタンは、チェックが入っていない状態と入っている状態という二つの状態を取ることのできるウィジェットです。チェックボタンに設定された Tk 変数オブジェクトの内容は、チェックが入っていないときは 0 という文字列で、チェックが入っているときは 1 という文字列です。

チェックボタンは、 TkCheckButton というクラスのインスタンスです。 text メソッドでチェックボタンに文字列を設定すると、チェックが表示される場所の右側に、その文字列が表示されます。

プログラムの例 check.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
var = TkVariable.new("0")
TkLabel.new do
  textvariable(var)
  font(f)
  pack
end
TkCheckButton.new do
  text("チェックボタン")
  variable(var)
  font(f)
  pack
end
Tk.mainloop
```

6.2.4 ラジオボタン

ラジオボタンも、チェックボタンと同じように、チェックが入っていない状態と入っている状態という二つの状態を持つウィジェットです。チェックボタンとラジオボタンとの相違点は、前者が単独で動作するのに対して、後者はグループで動作するというところにあります。

ラジオボタンのグループは、ひとつの Tk 変数オブジェクトを使って作ることができます。同一の Tk 変数オブジェクトを複数のラジオボタンに設定すると、それらのラジオボタンはひとつのグループを形成します。

ラジオボタンのひとつのグループの中では、チェックが入っているのは常にひとつだけです。そして、チェックが入っていないラジオボタンをクリックすると、クリックしたほうにチェックが移動します。この動作は、次のようなメカニズムによって実現されています。

ラジオボタンは、自分を識別するために 1 個の文字列を保持しています。実は、ラジオボタンにチェックが入るのは、Tk 変数オブジェクトが保持している文字列と、自分を識別する文字列とが一致しているときだけなのです。そしてラジオボタンは、自分がクリックされたときに自分を識別する文字列を Tk 変数オブジェクトに設定します。ですから、チェックが入っていないラジオボタンをクリックすると、クリックしたほうへチェックが移動することになります。

ちなみに、このメカニズムからわかるとおり、チェックが入っているラジオボタンがどれなのかということを知りたいときは、Tk 変数オブジェクトが保持している文字列を調べればいいわけです。

ラジオボタンは、 TkRadioButton というクラスのインスタンスです。チェックボックスと同

キー	値
side	ウィジェットを詰め込む方向
anchor	配置領域の中での位置
fill	引き伸ばしの指定
padx	x 軸方向の余白の大きさ
pady	y 軸方向の余白の大きさ
ipadx	x 軸方向の詰めものの大きさ
ipady	y 軸方向の詰めものの大きさ

表 6.2: pack が受け取るハッシュのキーと値

じように、text メソッドでラジオボタンに文字列を設定すると、チェックが表示される場所の右側に、その文字列が表示されます。

ラジオボタンを作るときには、それを識別するための文字列を設定する必要があります。その文字列は、ラジオボタンが持っている value というメソッドを使うことによって設定します。value は、引数として文字列を受け取って、それを、レシーバーを識別するための文字列として設定します。

プログラムの例 radio.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
gender = TkVariable.new("female")
TkLabel.new do
  textvariable(gender)
  font(f)
  pack
end
TkRadioButton.new do
  text("女性")
  value("female")
  variable(gender)
  font(f)
  pack
end
TkRadioButton.new do
  text("男性")
  value("male")
  variable(gender)
  font(f)
  pack
end
Tk.mainloop
```

6.3 pack

6.3.1 pack が受け取る引数

ウィジェットを土台の中に詰め込む pack というジオメトリーマネージャーには、どのようにウィジェットを詰め込むのかということを指定するためのハッシュを引数として渡すことができます。そのハッシュは、表 6.2 に示したようなキーと値から構成されます。

6.3.2 詰め込みの方向

pack を使ってウィジェットを思ったとおりの位置に配置するためには、そのウィジェットを土台に対してどういう方向へ詰め込むのかということを指定する必要があります。それを指定したいときは、pack に渡すハッシュの side というキーに対して値を与えます。

side に与えることのできる値は、top、bottom、left、right のいずれかで、それらの値は、土台に対してウィジェットを詰め込む方向をあらわしています。デフォルトは top です。

packによってウィジェットがどのように配置されのるかというのは、詰め込む順番に依存して決まります。指定された方向へウィジェットが詰め込まれると、それとは逆の方向に、仮想的な残りの領域ができます。そして、それ以降に詰め込まれるウィジェットは、その残りの領域を使うことになります。ですから、ウィジェットを詰め込む方向というのは、それ以降に詰め込むウィジェットの位置を決定するためのものだと考えるとわかりやすいでしょう。

プログラムの例 side.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
TkButton.new do
  text("1:left")
  font(f)
  pack("side"=>"left")
end
TkButton.new do
  text("2:top")
  font(f)
  pack("side"=>"top")
end
TkButton.new do
  text("3:left")
  font(f)
  pack("side"=>"left")
end
Tk.mainloop
```

このプログラムでは、ひとつ目のウィジェットを左の方向へ詰め込んでいます。ですから、それ以降に詰め込むウィジェットは、その右側の領域を使うことになります。そして、二つ目のウィジェットは上の方向へ詰め込まれていますので、三つ目のウィジェットはその下に配置されます。

6.3.3 配置領域の中での位置

土台の上に残されている、ウィジェットを詰め込むことのできる領域のことを、「配置領域」と呼ぶことにしましょう。そして、詰め込まれるウィジェットそのものの領域のことを「表示領域」と呼ぶことにしましょう。

表示領域が配置領域よりも小さい場合、ウィジェットは、デフォルトでは配置領域の中央に配置されます。しかし、配置領域の中のそれ以外の位置にウィジェットを配置することも可能です。それをしたいときは、 pack に渡すハッシュの anchor というキーに値を与えます。

anchor に与えることのできる値は、 center 、 n 、 ne 、 e 、 se 、 s 、 sw 、 w 、 nw のいずれかで、 center 以外のそれぞれの値は、北、北東、東、南東、などの方位をあらわしています。

プログラムの例 anchor.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
TkButton.new do
  text("左右に長いウィジェット")
  font(f)
  pack
end
TkButton.new do
  text("右寄せ")
  font(f)
  pack("anchor"=>"e")
end
Tk.mainloop
```

6.3.4 引き延ばし

ウィジェットの大きさというのは、基本的には、その内部に表示されるものの大きさによって決まります。しかし、いくつかのウィジェットを並べて配置する場合には、それらの大きさをそろえないと、見栄えがよくありません。

並んでいるいくつかのウィジェットの大きさをそろえたいときは、小さなウィジェットを大きく引き延ばす、という技法を使います。ウィジェットの大きさを引き延ばしたいときは、 pack

に渡すハッシュの fill というキーに値を与えます。

fill に与えることのできる値は、none、x、y、both のいずれかです。none は引き延ばさない、x は x 軸方向にのみ引き延ばす、y は y 軸方向にのみ引き延ばす、both はどちらの方向にも引き延ばす、という意味で、デフォルトは none です。

プログラムの例 fill.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
TkButton.new do
  text("左右に長いウィジェット")
  font(f)
  pack
end
TkButton.new do
  text("短い")
  font(f)
  pack("fill"=>"x")
end
Tk.mainloop
```

6.3.5 余白

ウィジェットの周囲に余白を作りたいときは、pack に渡すハッシュの padx または pady というキーに値を与えます。padx に与える値は x 軸方向の余白の大きさ、pady に与える値は y 軸方向の余白の大きさで、単位はピクセルです。

プログラムの例 pad.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
TkButton.new do
  text("デフォルト")
  font(f)
  pack
end
TkButton.new do
  text("上下左右に余白")
  font(f)
  pack("padx"=>50, "pady"=>50)
end
Tk.mainloop
```

6.3.6 詰めもの

ウィジェットの上に表示されるテキストなどの周囲に詰めものを詰めることによってウィジェットの大きさを大きくする、ということも可能です。それをしたいときは、pack に渡すハッシュの ipadx または ipady というキーに値を与えます。ipadx は x 軸方向の詰めものの大きさ、ipady は y 軸方向の詰めものの大きさで、単位はピクセルです。

プログラムの例 ipad.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
TkButton.new do
  text("デフォルト")
  font(f)
  pack
end
TkButton.new do
  text("上下左右に詰めもの")
  font(f)
  pack("ipadx"=>50, "ipady"=>50)
end
Tk.mainloop
```

キー	値
type	メッセージボックスの種類
icon	表示するアイコン
default	デフォルトにするボタンの名前
title	タイトルバーに表示する文字列
message	表示する文字列

表 6.3: messageBox が受け取るハッシュのキーと値

6.4 ダイアログボックス

6.4.1 ダイアログボックスの基礎

人間との対話のためにプログラムが一時的に画面に表示するウィンドウは、「ダイアログボックス」(dialog box) と呼ばれます。

いくつかの定型的なダイアログボックスは、Tk というモジュールが持っているメソッドを使うことによって表示することができます。

6.4.2 メッセージボックス

人間に対してメッセージを伝えたり、人間に対して単純な問い合わせをしたりするときに使われるダイアログボックスは、「メッセージボックス」(message box) と呼ばれます。

メッセージボックスを表示したいときは、Tk が持っている messageBox というメソッドを使います。

messageBox は、表示するメッセージボックスを細かく設定するためのハッシュを引数として受け取ります。そのハッシュは、表 6.3 に示されているキーと値から構成されます。

type というキーに対応する値は、メッセージボックスの種類をあらわす文字列です。メッセージボックスの種類としては、

ok okcancel yesno yesnocancel retrycancel abortretryignore

という 6 種類のものがあります。デフォルトは ok です。

icon というキーに対応する値は、表示するアイコンをあらわす文字列です。アイコンは、info、question、error という 3 種類の中から選ぶことができます。デフォルトは info です。

default というキーに対応する値は、デフォルトにするボタンの名前です。メッセージボックスのそれぞれのボタンには、ok、cancel、yes、no というような名前が付いていますので、その名前でボタンを指定します。

プログラムの例 message.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
TkButton.new do
  text("押してください。")
  font(f)
  command do
    Tk.messageBox("title" => "thanks",
      "message" => "ありがとうございました。")
  end
  pack
end
Tk.mainloop
```

messageBox は、戻り値として、押されたボタンの名前を返します。

プログラムの例 yesno.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
returned = TkLabel.new do
  font(f)
```

キー	値
filetypes	選択可能なファイルのタイプ
defaultextension	最初に選択されている拡張子
initialdir	最初に選択されているディレクトリー
initialfile	最初に入力されているファイル名
title	タイトルバーに表示する文字列

表 6.4: ファイルを選択するダイアログボックスが受け取るハッシュのキーと値

```
  pack
end
TkButton.new do
  text("押してください。")
  font(f)
  command do
    returned.text("")
    returned.text(Tk.messageBox("type" => "yesno",
      "icon" => "question", "default" => "no",
      "title" => "emptiness",
      "message" => "お腹が空きましたか？"))
  end
  pack
end
Tk.mainloop
```

6.4.3 ファイルを選択するダイアログボックス

Tk が持っている getOpenFile または getSaveFile というメソッドを使うことによって、ファイルを選択するためのダイアログボックスを表示することができます。

getOpenFile はデータを読み込むファイルを選択するためのメソッドで、存在しないファイルを選択した場合にエラーのダイアログボックスを表示します。それに対して、getSaveFile のほうはデータを保存するファイルを選択するためのメソッドで、すでに存在するファイルを選択した場合、上書きしてもいいかどうかを確認するダイアログボックスを表示します。

getOpenFile と getSaveFile も、ひとつのハッシュを引数として受け取ります。そのハッシュは、表 6.4 に示されているキーと値から構成されます。

filetypes というキーに与える値は、ファイルのタイプ名と拡張子から構成される配列を並べてできる、

```
[["css", ".css"], ["html", [".htm", ".html"]], ["all", ".*"]]
```

というような配列です。

getOpenFile と getSaveFile は、戻り値として、選択されたファイルの絶対パス名を返します。ただし、ダイアログボックスをキャンセルで閉じた場合は空文字列を返します。

プログラムの例 getopen.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
returned = TkLabel.new do
  font(f)
  pack
end
TkButton.new do
  text("ファイルの選択")
  font(f)
  command do
    returned.text("")
    returned.text(Tk.getOpenFile(
      "filetypes" => [
        ["ruby", ".rb"],
        ["C", ".c"],
```

```
        ["all", ".*"]],
      "defaultextension" => ".rb"))
  end
  pack
end
Tk.mainloop
```

6.4.4 色を選択するダイアログボックス

色を選択するためのダイアログボックスを表示したいときは、Tk が持っている chooseColor というメソッドを使います。このメソッドは、選択された色をあらわす文字列を戻り値として返します（ダイアログボックスをキャンセルで閉じた場合は空文字列を返します）。

chooseColor も、ひとつのハッシュを引数として受け取ります。initialcolor というキーに、色をあらわす文字列を値として与えることによって、選択される色の初期値を設定することができます。

プログラムの例 chcolor.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
color = "#99ccff"
returned = TkLabel.new do
  font(f)
  text(color)
  background(color)
  pack("ipadx"=>50, "ipady"=>50)
end
TkButton.new do
  text("色の選択")
  font(f)
  command do
    newcolor = Tk.chooseColor("initialcolor"=>color)
    if newcolor != ""
      color = newcolor
      returned.text(color)
      returned.background(color)
    end
  end
  pack
end
Tk.mainloop
```

6.5 メニューバー

6.5.1 メニューバーの基礎

メニューを表示するための帯状のウィジェットは、「メニューバー」(menubar) と呼ばれます。メニューバーは、TkMenubar というクラスのインスタンスです。

メニューバーを生成するためには、メニューの仕様を記述した配列を作る必要があります。そして、その配列を作るためには、「手続きオブジェクト」と呼ばれるものを作る必要があります。

というわけで、メニューバーの生成について説明するのに先立って、まず、手続きオブジェクトについて説明して、次に、メニューの仕様を記述する方法について説明することにしましょう。

6.5.2 手続きオブジェクト

「手続きオブジェクト」(procedure object) というのは、何らかの動作をあらわしているオブジェクトのことです。手続きオブジェクトは、Proc というクラスのインスタンスです。

手続きオブジェクトは、proc という関数的メソッドを使うことによって生成することができます。proc は、受け取ったブロックを実行するという動作をあらわす手続きオブジェクトを生成して、それを戻り値として返します。たとえば、

```
sanbai = proc { |x| x * 3 }
```

という式を評価すると、受け取ったオブジェクトを 3 倍するという動作をあらわす手続きオブ

ジェクトが生成されて、それが sanbai という変数に代入されます。

手続きオブジェクトがあらわしている動作を実行したいときは、それが持っている call というメソッドを呼び出します。call に引数を渡すと、その引数は、手続きオブジェクトによって実行されるブロックに渡されます。そして、ブロックの中の式の値が、call の戻り値になります。たとえば、sanbai という変数に先ほどの手続きオブジェクトが代入されているとするとき、

```
sanbai.call(7)
```

というメッセージ式を評価すると、21 という整数が値として得られます。

6.5.3 メニューの仕様

メニューバーを生成するために必要となる、メニューの仕様を記述した配列というのは、配列の配列の配列、つまり三重の入れ子になった配列です。

メニューの仕様を記述した配列のそれぞれの要素は、

```
[[項目名],
  [項目名, 手続きオブジェクト],
  ⋮
]
```

という形の配列です。この配列で、メニューバーを構成するひとつの項目の仕様を記述します。先頭の要素は、メニューバーの上に最初から表示されている項目をあらわしていて、それに続く要素は、メニューバーの項目がクリックされたときに表示されるメニューの項目をあらわしています。「項目名」のところには、メニューの上に表示される文字列を書き、「手続きオブジェクト」のところには、項目がクリックされたときに実行される動作をあらわす手続きオブジェクトを書きます。たとえば、

```
[["うどん"],
  ["きつね", proc { kitsune }],
  ["月見",   proc { tsukimi }],
  ["カレー", proc { curry }]
]
```

という配列を含んでいる配列からメニューバーを生成すると、そのメニューバーの上に「うどん」という項目が表示され、その項目をクリックすると、「きつね」と「月見」と「カレー」という項目から構成されるメニューが表示されます。そして、そのメニューの項目のいずれかをクリックすると、それに対応する手続きオブジェクトが実行されます。

6.5.4 メニューバーの生成

メニューバーは、ほかのウィジェットと同じように、new というクラスメソッドを使って生成します。メニューバーを生成する TkMenubar というクラスの new には、二つの引数を渡す必要があります。1 個目の引数は nil で、2 個目の引数はメニューバーの仕様を記述した配列です。

メニューバーは、tearoff というメソッドを持っています。これは、メニューバーからメニューを切り離すことができるかどうかを設定するメソッドです。引数として true を渡すと切り離しができるようになり、false を渡すと切り離しができなくなります。デフォルトは true です。

プログラムの例 menubar.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
def order(item)
  Tk.messageBox("message" => "注文: " + item)
end
spec = [
  [["料理"],
    ["焼きそば", proc { order("焼きそば") }],
    ["カレー",   proc { order("カレー") }]
  ],
  [["飲み物"],
    ["烏龍茶",   proc { order("烏龍茶") }],
    ["コーヒー", proc { order("コーヒー") }]
```

```
  ]
]
TkMenubar.new(nil, spec) do
  tearoff(false)
  font(f)
  pack
end
Tk.mainloop
```

6.6 入力のウィジェット

6.6.1 エントリー

改行を含まない文字列を人間から受け取るウィジェットは、「エントリー」(entry) と呼ばれます。エントリーは、TkEntry というクラスのインスタンスです。

エントリーの横の長さを設定したいときは、width というメソッドを使います。width は、表示することのできる文字数を引数として受け取ります。なお、入力することのできる文字列の長さは、エントリーの横の長さには制限されません。

エントリーに入力されている文字列を取り出したいときは、value というメソッドを使います。value は、エントリーに入力されている文字列を戻り値として返します。また、value= というメソッドを使うことによって、エントリーに文字列を設定することも可能です。

6.6.2 フォーカス

キーボードというのはひとつのコンピュータにひとつだけしかありませんので、表示されているいくつかのウィジェットのうちで、キーボードから入力された文字を受け取ることができるものは、ひとつの時点ではひとつだけです。キーボードから入力された文字を受け取ることができる状態にあるウィジェットは、「フォーカス」(focus) が設定されている、と言われます。

フォーカスの設定は、マウスやキーボードの操作によって切り替えることも可能ですが、プログラムの側で特定のウィジェットにフォーカスを設定することも可能です。それをしたいときは、フォーカスを設定したいウィジェットが持っている focus というメソッドを呼び出します。

プログラムの例 entry.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
label = TkLabel.new do
  font(f)
  pack
end
entry = TkEntry.new do
  width(30)
  font(f)
  focus
  pack
end
TkButton.new do
  text("転送")
  font(f)
  command do
    label.text(entry.value)
  end
  pack
end
Tk.mainloop
```

textvariable というメソッドを使って Tk 変数オブジェクトをエントリーに設定すると、エントリーに入力された文字列は、その Tk 変数オブジェクトに格納されることになります。

プログラムの例 entry2.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
var = TkVariable.new("")
TkLabel.new do
```

```
  textvariable(var)
  font(f)
  pack
end
TkEntry.new do
  width(30)
  textvariable(var)
  font(f)
  focus
  pack
end
Tk.mainloop
```

6.6.3 テキストウィジェット

改行を含む文字列、つまりいくつかの行から構成される文字列を人間から受け取るウィジェットは、「テキストウィジェット」(text widget) と呼ばれます。テキストウィジェットは、TkText というクラスのインスタンスです。

テキストウィジェットの大きさを設定したいときは、height と width というメソッドを使います。height は表示することのできる行数を設定するメソッドで、width は 1 行に表示することのできる文字数を設定するメソッドです。

テキストウィジェットの横の長さよりも長い行をどう表示するかということについては、デフォルトでは、行の途中で折り返して表示するという設定になっています。行を折り返さずに、左右にスクロールできるように設定したいときは、wrap というメソッドを呼び出して、引数として none という文字列を渡します。

プログラムの例 text.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
TkText.new do
  height(10)
  width(30)
  wrap("none")
  font(f)
  focus
  pack
end
Tk.mainloop
```

6.6.4 スクロールバー

ウィジェットをスクロールさせるために使われる細長いウィジェットは、「スクロールバー」(scrollbar) と呼ばれます。スクロールバーは、TkScrollbar というクラスのインスタンスです。

スクロールバーの操作によってテキストウィジェットの表示がスクロールするようにしたいときは、テキストウィジェットが持っている、xscrollbar または yscrollbar というメソッドを使います。これらのメソッドは、引数としてスクロールバーを受け取って、それをレシーバーに設定します。xscrollbar は水平方向のスクロールに使うスクロールバーを設定し、yscrollbar は垂直方向のスクロールに使うスクロールバーを設定します。

プログラムの例 scroll.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>20)
scroll = TkScrollbar.new do
  pack("side"=>"right", "fill"=>"y")
end
TkText.new do
  height(10)
  width(30)
  font(f)
  focus
  yscrollbar(scroll)
  pack("side"=>"right", "fill"=>"y")
end
```

Tk.mainloop

6.6.5 テキストエディター

テキストウィジェットを使って、ごく単純なテキストエディターを書いてみましょう。

テキストウィジェットに入力された文字列を取り出したり、テキストウィジェットに文字列を設定したりしたいときは、エントリーの場合と同じように、 value と value= というメソッドを使います。

プログラムの例 editor.rb

```
require("tk")

class Editor
  def initialize(path)
    f = TkFont.new("family"=>"times", "size"=>20)
    spec = [
      [["ファイル"],
        ["開く...",             proc { open }],
        ["保存",                proc { save }],
        ["名前を付けて保存...", proc { saveAs }]
      ]
    ]
    TkMenubar.new(nil, spec) do
      tearoff(false)
      font(f)
      pack("side"=>"top", "anchor"=>"w")
    end
    scroll = TkScrollbar.new do
      pack("side"=>"right", "fill"=>"y")
    end
    @text = TkText.new do
      width(70)
      height(20)
      font(f)
      focus
      yscrollbar(scroll)
      pack("side"=>"right", "fill"=>"y")
    end
    @path = ""
    readFile(path)
    Tk.mainloop
  end
  def readFile(path)
    if path != ""
      File.open(path) do |f|
        @text.value = f.read
      end
      @path = path
    end
  end
  def writeFile(path)
    if path != ""
      File.open(path, "w") do |f|
        f.write(@text.value)
      end
      @path = path
    end
  end
  def open
    readFile(Tk.getOpenFile)
  end
  def save
    if @path != ""
      writeFile(@path)
    else
      saveAs
```

クラス名	説明
TkcRectangle	長方形
TkcOval	楕円
TkcArc	円弧
TkcLine	折れ線
TkcPolygon	多角形
TkcText	文字列
TkcWindow	ウィジェット
TkcBitmap	XBM 形式の画像
TkcImage	PPM、PGM、GIF などの形式の画像

表 6.5: キャンバスオブジェクトのクラス

```
    end
  end
  def saveAs
    writeFile(Tk.getSaveFile)
  end
end

if ARGV.size == 0
  Editor.new("")
else
  Editor.new(ARGV[0])
end
```

6.7 キャンバス

6.7.1 キャンバスの生成

Tk には、「キャンバス」(canvas) と呼ばれるウィジェットがあります。これは、その上に図形を描画することができるウィジェットで、 TkCanvas というクラスのインスタンスです。

キャンバスの大きさは、 height と width というメソッドを使って設定します。 height が縦の長さ、 width が横の長さで、単位はピクセルです。

キャンバスの背景の色は、ほかのウィジェットと同じように、 background というメソッドを使って設定します。

プログラムの例 canvas.rb

```
require("tk")
TkCanvas.new do
  height(200)
  width(300)
  background("#ccffff")
  pack
end
Tk.mainloop
```

図形や文字列やウィジェットなど、キャンバスの上に表示されるものは、「キャンバスオブジェクト」(canvas object) と呼ばれます。

キャンバスの上にキャンバスオブジェクトを表示するためには、それを表示するオブジェクトを生成する必要があります。キャンバスオブジェクトを表示するオブジェクトも、「キャンバスオブジェクト」と呼ばれます。表 6.5 は、キャンバスオブジェクトを生成するクラスの一覧です。

キャンバスオブジェクトを生成するためには、そのクラスの new を呼び出して、いくつかの引数を渡す必要があります。1 個目の引数はキャンバスで、2 個目以降は、キャンバス上の位置を指定する座標などです。

たとえば、長方形を表示したいときは、 TkcRectangle の new に対して、

キャンバス, x_1, y_1, x_2, y_2

という引数を渡します。(x_1, y_1) と (x_2, y_2) は、長方形の対角線の両端の座標です。ちなみに、キャンバスの座標系は、左上の隅が原点で、x 軸は右向き、y 軸は下向きです。

プログラムの例 rect.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcRectangle.new(canvas, 50, 100, 300, 200)
TkcRectangle.new(canvas, 200, 50, 350, 250)
Tk.mainloop
```

図形のキャンバスオブジェクトは、デフォルトでは、線の太さが 1 ピクセル、線の色が黒で、内部を塗りつぶさずに表示されます。

線の太さ、線の色、内部を塗りつぶす色を設定したいときは、new にブロックを渡して、そのブロックの中でメソッドを呼び出します。線の太さを設定するメソッドは width、線の色を設定するメソッドは outline、そして図形の内部の色を設定するメソッドは fill です。width は太さをあらわす整数を引数として受け取り、outline と fill は、色をあらわす文字列を引数として受け取ります。outline に空文字列を渡すと線が表示されなくなり、fill に空文字列を渡すと内部が塗りつぶされなくなります。

プログラムの例 initobj.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcRectangle.new(canvas, 50, 50, 300, 200) do
  width(10)
  outline("#0080ff")
  fill("#80ff80")
end
TkcRectangle.new(canvas, 100, 100, 350, 250) do
  outline("")
  fill("#ffff80")
end
Tk.mainloop
```

6.7.2 楕円と円弧

キャンバスの上に楕円を表示したいときは、TkcOval というクラスのインスタンスを生成します。TkcOval の new には、

キャンバス, x_1, y_1, x_2, y_2

という引数を渡します。(x_1, y_1) と (x_2, y_2) は、楕円に外接する長方形の対角線の両端の座標です。

プログラムの例 oval.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcOval.new(canvas, 50, 100, 350, 200)
TkcOval.new(canvas, 100, 150, 200, 250)
TkcOval.new(canvas, 250, 50, 300, 250)
Tk.mainloop
```

メソッド名	説明
arrow	矢印の指定（none、first、last、both）
arrowshape	矢印の形（3 個の数値から構成される配列）
capstyle	線の端点の形状（butt、projecting、round）
joinstyle	線の接続点の形状（miter、bevel、round）

表 6.6: 折れ線の形状を設定するメソッド

キャンバスの上に円弧を表示したいときは、TkcArc というクラスのインスタンスを生成します。TkcArc の new に渡す引数は、楕円の場合と同じです。

円弧を表示するためには、new に渡すブロックの中で、start、extent、style というメソッドを呼び出すことによって、その円弧の属性を設定する必要があります。

start は、円弧の開始点の角度を設定するメソッドです。角度は、中心から見て右方向が 0 度で、反時計回りに大きくなっていきます。

extent は、円弧の終了点の角度を設定するメソッドです。プラスの数値を指定した場合は、開始点から終了点に向かって反時計回りに円弧が表示され、マイナスの数値を指定した場合は、時計回りに円弧が表示されます。

style は、円弧のスタイルを設定するメソッドです。arc という文字列を引数として渡すと、円弧のみが表示されます。chord を渡すと、開始点と終了点をつなぐ直線が追加されます。pieslice を渡すと、中心と開始点、中心と終了点をつなぐ直線が追加されます。

プログラムの例　arc.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcArc.new(canvas, 50, 30, 250, 200) do
  start(0)
  extent(225)
  style("arc")
end
TkcArc.new(canvas, 100, 80, 300, 250) do
  start(0)
  extent(225)
  style("chord")
end
TkcArc.new(canvas, 150, 130, 350, 300) do
  start(0)
  extent(225)
  style("pieslice")
end
Tk.mainloop
```

6.7.3 折れ線と多角形

キャンバスの上に折れ線を表示したいときは、TkcLine というクラスのインスタンスを生成します。TkcLine の new には、

キャンバス, x_1, y_1, x_2, y_2, $\cdots$, x_n, y_n

という引数を渡します。そうすると、(x_1, y_1)、(x_2, y_2)、$\cdots$、(x_n, y_n)、という点を順番に直線で連結した折れ線が描画されます。

折れ線の形状は、表 6.6 に示したメソッドを使うことによって設定することができます。ちなみに、arrowshape に渡す引数は 1 個の配列で、1 個目は矢印の根元から先端までの長さ、2 個目は翼端をつなぐ直線と中心線との交点から先端までの長さ、3 個目は中心線と翼端とのあいだの距離です。

プログラムの例 line.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcLine.new(canvas, 30, 30, 150, 150, 30, 270)
TkcLine.new(canvas, 100, 30, 220, 150, 100, 270) do
  arrow("last")
  arrowshape([30, 40, 10])
end
TkcLine.new(canvas, 170, 30, 290, 150, 170, 270) do
  width(20)
  capstyle("round")
end
TkcLine.new(canvas, 240, 30, 360, 150, 240, 270) do
  width(20)
  joinstyle("bevel")
end
Tk.mainloop
```

キャンバスの上に多角形を表示したいときは、TkcPolygon というクラスのインスタンスを生成します。TkcPolygon の new には、多角形を構成するそれぞれの頂点の座標を引数として渡します。

プログラムの例 polygon.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcPolygon.new(canvas, 200, 100, 30, 200, 370, 200) do
  outline("")
  fill("#80ff00")
end
TkcPolygon.new(canvas,
    120, 50, 80, 250, 320, 250, 280, 50) do
  width(20)
  outline("#8000ff")
  fill("")
end
Tk.mainloop
```

6.7.4 画像

Ruby/Tk で扱うことのできる画像は、「ビットマップイメージ」(bitmap image) と「フォトイメージ」(photo image) という 2 種類のものに分類することができます。ビットマップイメージのほうは単色の画像で、フォトイメージのほうはフルカラーの画像です。

フォトイメージは、TkPhotoImage というクラスのインスタンスとして扱われます。このクラスのインスタンスに対して画像のデータを設定したいときは、そのオブジェクトが持っている file というメソッドを使います。このメソッドを呼び出して、画像のデータが格納されているファイルのパス名を引数として渡すと、このメソッドは、ファイルから画像のデータを読み込んで、それをレシーバーに設定します。たとえば、

```
TkPhotoImage.new do
  file("namako.gif")
end
```

という式を書くことによって、TkPhotoImage クラスのインスタンスを生成して、namako.gif というパス名で指定されたファイルに格納されている画像データをそれに設定することができます。

キャンバスの上にフォトイメージを表示したいときは、TkcImage というクラスのインスタンスを生成します。TkcImage の new には、

キャンバス，x，y

という引数を渡します。そうすると、(x, y) という点を中心とする位置に画像を表示するキャンバスオブジェクトが生成されます。

TkcImage クラスのインスタンスに画像を表示させるためには、そのキャンバスオブジェクトが持っている image というメソッドを使って、フォトイメージの設定をする必要があります。image は、フォトイメージを引数として受け取って、それをレシーバーに設定します。

プログラムの例 image.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
img = TkPhotoImage.new do
  file("sample.gif")
end
TkcImage.new(canvas, 200, 150) do
  image(img)
end
Tk.mainloop
```

6.7.5 キャンバスオブジェクトの移動

すべてのキャンバスオブジェクトは、move というメソッドを持っています。これは、レシーバーが表示されている位置を移動させるメソッドです。

move は、x 軸方向の距離と y 軸方向の距離を指定する 2 個の整数（単位はピクセル）を引数として受け取って、それらの距離だけレシーバーを移動させます。

プログラムの例 move.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
circle = TkcOval.new(canvas, 190, 140, 220, 170) do
  fill("#ffcc00")
end
TkButton.new do
  text("右")
  font(f)
  command do
    circle.move(10, 0)
  end
  pack("side"=>"right")
end
TkButton.new do
  text("左")
  font(f)
  command do
    circle.move(-10, 0)
  end
  pack("side"=>"left")
end
Tk.mainloop
```

6.8 トップレベル

6.8.1 トップレベルの生成

第 6.1 節で説明したように、ウィンドウを生成するクラスとしては、TkRoot と TkToplevel という二つのものがあります。TkRoot のインスタンスは、ルートウィジェット、つまり mainloop を呼び出したときに自動的に表示されるウィンドウです。そして、TkToplevel のインスタンスは、「トップレベル」(toplevel) と呼ばれるウィンドウです。

TkToplevel クラスのインスタンスを new を使って生成すると、それだけでウィンドウが表示されます。タイトルバーに表示する文字列は、ルートウィジェットの場合と同じように、title というメソッドを呼び出すことによって設定することができます。

プログラムの例 toplevel.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
count = 0
TkButton.new do
  text("ウィンドウを開く")
  font(f)
  command do
    count += 1
    TkToplevel.new do
      title("window " + count.to_s)
    end
  end
  pack
end
Tk.mainloop
```

6.8.2 トップレベルへのウィジェットの取り付け

ウィジェットを生成するクラスの new は、そのウィジェットを取り付ける土台となるウィジェットを 1 個目の引数として受け取ります。ですから、ウィジェットを生成するクラスの new を呼び出すときに、1 個目の引数としてトップレベルを渡すことによって、そのトップレベルの上にウィジェットを取り付けることができます。

ちなみに、1 個目の引数として nil を渡すか、または引数をすべて省略した場合は、土台としてルートウィジェットを指定するという意味になります。

プログラムの例 countup2.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>18)
count = 0
countlabel = TkLabel.new do
  text("0")
  font(f)
  pack("padx" => 50)
end
toplevel = TkToplevel.new
TkButton.new(toplevel) do
  text("増やす")
  font(f)
  command do
    count += 1
    countlabel.text(count.to_s)
  end
  pack
end
Tk.mainloop
```

6.8.3 グラブ

トップレベルを使うことによって、独自の機能を持つダイアログボックスを作ることができます。ただし、独自のダイアログボックスを作るためには、ウィジェットが持っているいくつかの特殊な機能を使う必要があります。その場合に必要になる機能というのは、グラブ、消滅、そし

て待機です。

「グラブ」(grab) というのは、自分に対する操作が終了するまでのあいだ、自分以外のウィジェットに対する操作を制限するという機能のことです。グラブを使いたいときは、ウィジェットが持っている grab というメソッドを呼び出します。

引数を渡さずに grab を呼び出した場合、grab は、ほかのプログラムに対する操作までは制限しません。ほかのプログラムに対しても操作を制限したいときは、global という文字列を引数として渡して grab を呼び出します。

グラブを解除したいときは、release という文字列を引数として渡して grab を呼び出します。

6.8.4 ウィジェットの消滅とその待機

ダイアログボックスは、自分に対する操作が終了したときに、自分自身を消滅させる必要があります。ウィジェットを消滅させたいときは、それが持っている destroy というメソッドを使います。

また、ダイアログボックスを表示するという処理は、そのダイアログボックスに対する操作が終了するまでのあいだ、その先へ進まずに待機している必要があります。処理を待機させたいときは、ウィジェットが持っている wait_destroy というメソッドを使います。このメソッドは、レシーバーが存在しているあいだは何もしないで待機していて、レシーバーが消滅すると終了します。

6.8.5 文字列を読み込むダイアログボックス

これまでに説明した grab や destroy や focus などを使えば、独自のダイアログボックスを作ることができます。

それでは、例として、文字列を読み込むダイアログボックスを作ってみましょう。

プログラムの例 getstr.rb

```
require("tk")

module Dialog
  def Dialog.getString(init, size)
    var = TkVariable.new(init)
    toplevel = TkToplevel.new do
      grab
    end
    f = TkFont.new("family"=>"helvetica", "size"=>18)
    entry = TkEntry.new(toplevel) do
      width(size)
      font(f)
      textvariable(var)
      focus
      pack
    end
    TkButton.new(toplevel) do
      text("OK")
      font(f)
      command do
        toplevel.grab("release")
        toplevel.destroy
      end
      pack
    end
    toplevel.wait_destroy
    var.value
  end
end

f = TkFont.new("family"=>"times", "size"=>18)
var = TkVariable.new("string")
label = TkLabel.new do
  textvariable(var)
  font(f)
  pack
end
```

タイプ	説明
ButtonPress	マウスのボタンを押す。
Button	ButtonPress と同じ。
ButtonRelease	マウスのボタンを離す。
Motion	マウスを移動させる。
Enter	マウスポインターを重ねる。
Leave	マウスポインターを離す。
KeyPress	キーボードのキーを押す。
Key	KeyPress と同じ。
KeyRelease	キーボードのキーを離す。

表 6.7: イベントのタイプ

```
TkButton.new do
  text("文字列の変更")
  font(f)
  command do
    var.value = Dialog.getString(var.value, 30)
  end
  pack
end
Tk.mainloop
```

6.9 バインディング

6.9.1 バインディングの基礎

GUI を持つプログラムは、それに対して人間が何らかの操作をしたときに、それに対応する処理を実行します。GUI の上で何らかの操作が発生した、という出来事は、「イベント」(event) と呼ばれます。そして、イベントが発生したときに実行される処理は、「イベント処理」(event processing) と呼ばれます。

イベントに対して何らかの動作をイベント処理として割り当てることを、イベントに動作を「バインドする」(bind) と言います（名詞形は「バインディング」(binding) です）。ウィジェットやキャンバスオブジェクトは、イベントが自分の上で発生したときに、そのイベントにバインドされている動作を実行することができる、という機能を持っています。

イベントに動作をバインドしたいときは、その動作を実行するウィジェットまたはキャンバスオブジェクトが持っている bind というメソッドを呼び出します。bind は、2 個または 3 個の引数を受け取ります。1 個目の引数はイベントの種類を指定する文字列で、2 個目の引数は、そのイベントにバインドする動作をあらわす手続きオブジェクトです。3 個目の引数については、もう少しあとで説明します。

イベントの種類を指定する文字列は、

修飾子 - タイプ - 詳細

という形式で書きます。ただし、最低限必要なのは「タイプ」という部分だけです。「タイプ」の部分には、表 6.7 に示されている、イベントのタイプを指定する文字列のいずれかを書きます。たとえば、ButtonPress という文字列を書くことによって、マウスのボタンを押すというイベントを指定することができます。

次のプログラムによって表示される数字は、マウスでクリックすると 1 だけ増えます。

プログラムの例 release.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>24)
count = 0
TkLabel.new do
  text("0")
```

```
  font(f)
  bind("ButtonRelease", proc do
    count += 1
    text(count.to_s)
  end)
  pack("padx" => 50)
end
Tk.mainloop
```

6.9.2 マウスのボタンの指定

マウスの特定のボタンによるイベントに限定して動作をバインドしたいときは、イベントのタイプを指定する文字列の右側にマイナスを書いて、そのさらに右側に、マウスのボタンを指定する番号を書きます。

3 個のボタンを持つマウスの場合、それぞれのボタンには、左から順番に、1、2、3、という番号が与えられています。ボタンが 2 個しかないマウスの場合は、左が 1 で右が 3 です。

次のプログラムによって表示される数字は、マウスの左ボタンでクリックすると 1 だけ増えて、右ボタンでクリックすると 1 だけ減ります。

プログラムの例 btnnum.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>24)
count = 0
TkLabel.new do
  text("0")
  font(f)
  bind("ButtonRelease-1", proc do
    count += 1
    text(count.to_s)
  end)
  bind("ButtonRelease-3", proc do
    count -= 1
    text(count.to_s)
  end)
  pack("padx" => 50)
end
Tk.mainloop
```

次のプログラムによって表示される長方形は、マウスの左ボタンでクリックすると線が太くなり、右ボタンでクリックすると線が細くなります。

プログラムの例 width.rb

```
require("tk")
w = 30
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcRectangle.new(canvas, 100, 100, 300, 200) do
  width(w)
  outline("#000080")
  fill("#00ff00")
  bind("ButtonRelease-1", proc do
    w += 2
    width(w)
  end)
  bind("ButtonRelease-3", proc do
    if w >= 2
      w -= 2
      width(w)
    end
  end)
end
Tk.mainloop
```

6.9.3 キーボードのキーの指定

キーボードの特定のキーによるイベントに限定して動作をバインドしたいときは、イベントのタイプを指定する文字列の右側にマイナスを書いて、そのさらに右側に、「キーシム」と呼ばれる文字列を書きます。

「キーシム」(keysym) というのは、キーボードのそれぞれのキーが持っている、自分を識別するための文字列のことです。たとえば、スペースは space、コンマは comma、スラッシュは slash、エンターは Enter、右のシフトは Shift_R、左向き矢印は Left、というキーシムで識別されます。なお、数字と英字のキーについては、そのキーの文字がそのままキーシムになっています。

次のプログラムによって表示される円は、矢印のキーを押すことによって上下左右に移動させることができます。

プログラムの例 keymove.rb

```
require("tk")
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
circle = TkcOval.new(canvas, 190, 140, 220, 170) do
  fill("#00ffcc")
end
TkRoot.new do
  bind("KeyPress-Right", proc do
    circle.move(10, 0)
  end)
  bind("KeyPress-Left", proc do
    circle.move(-10, 0)
  end)
  bind("KeyPress-Up", proc do
    circle.move(0, -10)
  end)
  bind("KeyPress-Down", proc do
    circle.move(0, 10)
  end)
end
Tk.mainloop
```

6.9.4 マウスの修飾子

イベントのタイプを指定する文字列の左側には、マイナスで区切って、「修飾子」(modifier) と呼ばれる文字列を何個でも好きなだけ書くことができます。修飾子を書くことによって、いくつかの操作の組み合わせであるようなイベントを指定することができます。

Button1 という文字列は、マウスの左ボタンを押しながら何らかの操作をするというイベントを指定するための修飾子です。たとえば、

```
Button1-Motion
```

という文字列を書くことによって、マウスの左ボタンでドラッグするというイベントを指定することができます。同様に、Button2 で真ん中のボタン、Button3 で右ボタンを指定することができます。

次のプログラムによって表示される数字は、マウスの左ボタンでドラッグすると、それに応じて増えていきます。

プログラムの例 drag.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>24)
count = 0
TkLabel.new do
  text("0")
  font(f)
  bind("Button1-Motion", proc do
    count += 1
```

文字列	説明
%b	マウスのボタンの番号
%x	ウィジェット内でのマウスの x 座標
%y	ウィジェット内でのマウスの y 座標
%X	画面全体でのマウスの x 座標
%Y	画面全体でのマウスの y 座標
%A	キーに対応する文字
%K	キーに対応するキーシム

表 6.8: イベントキーワード

```
    text(count.to_s)
  end)
  pack("padx" => 50)
end
Tk.mainloop
```

6.9.5 キーボードの修飾子

Shift、Control、Alt という修飾子を書くことによって、シフトキー、コントロールキー、オルトキーを押しながら何らかの操作をする、というイベントを指定することができます。

なお、シフトキーを押しながら別のキーを操作するというイベントを指定したいときは、修飾子を使うのではなくて、その操作に対応するキーシムを使う必要があります。たとえば、シフトキーを押しながら a のキーを押すというイベントは、KeyPress-A という文字列で指定されます。

次のプログラムによって表示される数字は、コントロールキーを押しながら上向き矢印キーを押すと、1 だけ増えます。

プログラムの例 control.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>24)
count = 0
label = TkLabel.new do
  text("0")
  font(f)
  pack("padx" => 50)
end
TkRoot.new do
  bind("Control-KeyPress-Up", proc do
    count += 1
    label.text(count.to_s)
  end)
end
Tk.mainloop
```

6.9.6 イベントキーワード

ウィジェットやキャンバスオブジェクトは、自分の上でイベントが発生したとき、発生したイベントに関する情報をあらわすオブジェクトを、そのイベントにバインドされている手続きオブジェクトに渡すことができます。ただし、そのためには、bind に渡す 3 個目の引数で、どのようなオブジェクトを渡すのかということを指定しておく必要があります。

手続きオブジェクトに渡すオブジェクトは、「イベントキーワード」(event keyword) と呼ばれる文字列を書くことによって指定します。イベントキーワードには、表 6.8 に示したようなものがあります。

イベントキーワードを空白で区切って並べることによってできた文字列を、3 個目の引数として bind に渡すと、その文字列に含まれるイベントキーワードによって指定されたオブジェクトが、イベントが発生したときに手続きオブジェクトに渡されます。たとえば、

```
%x %y
```

という文字列を bind に渡すと、イベントが発生したときのマウスの x 座標と y 座標が手続きオブジェクトに渡されます。

次のプログラムは、キャンバスの上でマウスが動かされたときに、そのときのマウスの座標をラベルで表示します。

プログラムの例 coordi.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>24)
label = TkLabel.new do
  font(f)
  pack
end
TkCanvas.new do
  width(400)
  height(300)
  background("#99ff99")
  bind("Motion", proc do |x, y|
    label.text("(" + x.to_s + "," + y.to_s + ")")
  end, "%x %y")
  pack
end
Tk.mainloop
```

次のプログラムは、キーボードのキーが押されたときに、押されたキーのキーシムをラベルで表示します。

プログラムの例 keysym.rb

```
require("tk")
f = TkFont.new("family"=>"times", "size"=>24)
label = TkLabel.new do
  font(f)
  pack("padx" => 50)
end
TkRoot.new do
  bind("KeyPress", proc do |keysym|
    label.text(keysym)
  end, "%K")
end
Tk.mainloop
```

次のプログラムによって表示される円は、マウスの左ボタンでドラッグすることによって移動させることができます。

プログラムの例 dragoval.rb

```
require("tk")
dx = 0
dy = 0
canvas = TkCanvas.new do
  width(400)
  height(300)
  pack
end
TkcOval.new(canvas, 190, 140, 220, 170) do
  fill("#99ff66")
  bind("ButtonPress", proc do |x, y|
    dx = x
    dy = y
  end, "%x %y")
  bind("Button1-Motion", proc do |x, y|
    move(x - dx, y - dy)
    dx = x
    dy = y
  end, "%x %y")
end
Tk.mainloop
```

6.10 タイマー

6.10.1 バックグラウンド処理

GUI を使って人間と対話をするプログラムは、基本的には、人間が何らかの操作をしたときだけ動作をします。つまり、人間が操作を何もしていないときは、自分も何もしていないわけです。しかし、GUI を持っていて、かつ、人間によって操作されていないあいだも何らかの処理を実行するようなプログラムを書きたい、という場合もあります。そのようなプログラムが実行する、人間による操作とは無関係に進行する処理は、「バックグラウンド処理」(background processing) と呼ばれます。

GUI を作るためのライブラリーの多くは、「タイマー」(timer) と呼ばれるものを作る機能を持っています。タイマーというのは、一定の時間ごとに動作を実行するオブジェクトのことです。バックグラウンド処理は、タイマーを使うことによって実現することができます。

6.10.2 タイマーの生成

Ruby/Tk では、タイマーは、`TkAfter` というクラスのインスタンスです。このクラスの `new` は、いくつかの引数を受け取って、それらの引数で指定されたタイマーを生成します。

`TkAfter` クラスの `new` が受け取る引数の 1 個目は、動作を実行する時間の間隔を指定する整数（単位はミリ秒）です。たとえば 5000 という整数を渡すと、5 秒間隔で動作を実行することになります。

引数の 2 個目は、動作を繰り返す回数を指定する整数です。無限に繰り返したい場合は、−1 を渡します。

引数の 3 個目は、実行する動作をあらわす手続きオブジェクトです。

たとえば、

```
TkAfter.new(10000, 4, proc { namako })
```

という式でタイマーを生成したとすると、そのタイマーは、`namako` というメソッドを 10 秒ごとに 4 回呼び出します。

手続きオブジェクトは、引数の 3 個目だけではなくて 4 個目以降にも好きなだけ引数として渡すことができます。2 個以上の手続きオブジェクトを渡した場合、それらの手続きオブジェクトは、一定の時間間隔ごとにローテーションで実行されることになります。

生成された直後のタイマーは、停止した状態になっています。タイマーに動作を開始させるためには、それが持っている `start` というメソッドを呼び出す必要があります。また、`stop` というメソッドを呼び出すことによって、動作しているタイマーを停止させることも可能です。

それでは、タイマーを使ってバックグラウンド処理を実行するプログラムを書いてみましょう。次のプログラムによって表示される数字は、1 秒ごとに 1 ずつ増加していきます。

プログラムの例 `timer.rb`

```
require("tk")
count = 0
f = TkFont.new("family"=>"times", "size"=>24)
label = TkLabel.new do
  text("0")
  font(f)
  pack("padx" => 50)
end
TkAfter.new(1000, -1, proc do
  count += 1
  label.text(count.to_s)
end).start
Tk.mainloop
```

6.11 ゲーム

6.11.1 ゲームの基礎

この節では、ゲームのプログラムというものはどうすれば書くことができるのか、ということについて考えてみたいと思います。

ゲームのプログラムが画面の上に表示するさまざまなグラフィックスのことを、「キャラクター」(character) と呼ぶことにします。大多数のゲームでは、キャラクターがゲームの進行にともなって移動していき、それらの位置関係によって得点が加算されたり勝敗が決まったりします。

次のプログラムは、キャラクターを表現するオブジェクトを生成する Character というクラスを定義しています。

プログラムの例 chara.rb

```
class Character
  def initialize(x, y, width, height)
    @x = x
    @y = y
    @width = width
    @height = height
  end
  def to_a
    [@x, @y, @width, @height]
  end
  def locate(x, y)
    @x = x
    @y = y
  end
  def move(mx, my)
    @x += mx
    @y += my
  end
  def outside(chara)
    x, y, width, height = chara.to_a
    if @x < x
      "west"
    elsif @x + @width > x + width
      "east"
    elsif @y < y
      "north"
    elsif @y + @height > y + height
      "south"
    else
      "inside"
    end
  end
  def Character.overlap(a, alength, b, blength)
    if a < b
      b <= a + alength
    else
      a <= b + blength
    end
  end
  def collision(chara)
    x, y, width, height = chara.to_a
    Character.overlap(@x, @width, x, width) &&
    Character.overlap(@y, @height, y, height)
  end
end
```

Character クラスは、キャラクターをひとつの長方形とみなして、その位置と大きさを扱います。キャラクターの位置と大きさは、@x、@y、@width、@height という 4 個のインスタンス変数によって示されます。@x と @y は長方形の左上の頂点の座標で、@width は横の長さ、@height は縦の長さです。

locate と move というメソッドは、キャラクターを移動させるためのものです。locate が移

動先の位置を絶対的な座標で受け取るのに対して、 move は、現在位置を基準とする相対的な座標で受け取ります。

outside というメソッドは、引数として 1 個のキャラクターを受け取って、レシーバーが完全に引数の内側にあるかどうかを判定します。完全に内側にある場合は inside という文字列を返し、そうでない場合は、外に出ている方角をあらわす文字列を返します。

キャラクターを実現する上で重要なのは、それが別のキャラクターと衝突したかどうかを判定するという処理です。そのような処理は、「当たり判定」(collision detection) と呼ばれます。

Character クラスの中で定義されている collision は、当たり判定をするメソッドです。このメソッドは、引数として 1 個のキャラクターを受け取って、レシーバーと引数とが少しでも重なっているかどうかを調べて、重なっているならば真、そうでなければ偽を返します。

Character クラスでは、キャラクターの形状を一律に長方形とみなして当たり判定をしますので、キャラクターの実際の形状が長方形ではない場合は、判定の結果に多少の誤差が生じます。本格的なゲームのプログラムを書く場合は、キャラクターの形状に応じた当たり判定のメソッドを書く必要があります。

6.11.2 テニスのプログラム

上で定義した Character クラスを使った具体的なゲームのプログラムの例として、テニスのプログラムを書いてみましょう。

このゲームは、プレーヤーがマウスを使ってラケットを動かすことによってボールを打ち返す、というものです。ラケットとボールは壁で囲まれたコートの中にあって、ボールは壁に当たると跳ね返ります[2]。得点は、ラケットでボールを打つたびに加算されます。そして、ラケットの背後にある壁にボールが当たった場合は、ゲームオーバーです。

それでは、次のプログラムを入力して、実行してみてください。

プログラムの例 tennis.rb

```
require("tk")
require("chara")

class Court < Character
  def initialize(cwidth, cheight, color)
    super(0, 0, cwidth, cheight)
    @canvas = TkCanvas.new do
      width(cwidth)
      height(cheight)
      background(color)
      pack
    end
  end
  def getCanvas
    @canvas
  end
end

class Racket < Character
  def initialize(width, height, court, color)
    cx, cy, cwidth, cheight = court.to_a
    @half_w = width / 2
    x = cwidth / 2 - @half_w
    y = cheight - (height + 50)
    super(x, y, width, height)
    @co = TkcRectangle.new(court.getCanvas,
            x, y, x + width, y + height) { fill(color) }
  end
  def locate(lx)
    lx -= @half_w
    x = to_a[0]
    move(lx - x, 0)
    @co.move(lx - x, 0)
  end
end
```

[2] どちらかと言えばテニスよりもスカッシュに近いですね。

```
class Ball < Character
  def initialize(size, court, color, velocity)
    super(0, 0, size, size)
    @co = TkcOval.new(court.getCanvas,
            0, 0, size, size) { fill(color) }
    @court = court
    @velocity = velocity
    reset
  end
  def reset
    @co.move(- to_a[0], - to_a[1])
    locate(0, 0)
    @direction = "se"
  end
  def turn
    case outside(@court)
    when "west"
      if @direction == "sw"
        @direction = "se"
      elsif @direction == "nw"
        @direction = "ne"
      end
    when "east"
      if @direction == "se"
        @direction = "sw"
      elsif @direction == "ne"
        @direction = "nw"
      end
    when "north"
      if @direction == "ne"
        @direction = "se"
      elsif @direction == "nw"
        @direction = "sw"
      end
    end
  end
  def beyondSouth
    outside(@court) == "south"
  end
  def move(x, y)
    super(@velocity * x, @velocity * y)
    @co.move(@velocity * x, @velocity * y)
  end
  def dirmove
    case @direction
    when "se"
      move(1, 1)
    when "sw"
      move(-1, 1)
    when "nw"
      move(-1, -1)
    when "ne"
      move(1, -1)
    end
  end
  def hit
    if @direction == "se"
      @direction = "ne"
    elsif @direction == "sw"
      @direction = "nw"
    end
  end
  def south
    @direction[0,1] == "s"
  end
```

```
end

class Tennis
  def initialize
    f = TkFont.new("family"=>"times", "size"=>24)
    @label = TkLabel.new do
      font(f)
      pack
    end
    court = Court.new(600, 500, "#006600")
    @racket = Racket.new(80, 20, court, "#ffffff")
    court.getCanvas.bind("Motion", proc do |x|
      @racket.locate(x)
    end, "%x")
    @ball = Ball.new(30, court, "#ffff99", 5)
    reset
    TkAfter.new(5, -1, proc { background }).start
    Tk.mainloop
  end
  def reset
    @score = 0
    @label.text("0")
    @ball.reset
  end
  def hit
    @ball.hit
    @score += 1
    @label.text(@score.to_s)
  end
  def background
    if @ball.beyondSouth
      Tk.messageBox("message" => "This game was over.")
      reset
    end
    @ball.turn
    if @ball.south && @ball.collision(@racket)
      hit
    end
    @ball.dirmove
  end
end

Tennis.new
```

6.11.3 テニスのキャラクター

Characterというのは、あくまでキャラクターというものを一般化したクラスですので、実際のキャラクターを生成するためには、Characterクラスのサブクラスを定義する必要があります。

テニスのゲームは、コート、ラケット、ボールという3個のキャラクターを持っています。それらのキャラクターは、それぞれ、Court、Racket、Ballというクラスから生成されます。そして、コートはキャンバスとして画面の上に表示され、ラケットとボールはキャンバスオブジェクトとしてキャンバスの上に表示されます。

ラケットは、マウスによって左右に移動させることができないといけませんので、locateという、そのためのメソッドを持っています。このメソッドは、引数として移動先の x 座標を受け取って、キャラクターとしての位置とキャンバスオブジェクトとしての位置の両方を移動させます。

ボールは、@directionというインスタンス変数を持っていて、その変数が指し示している文字列が、現在の進行方向をあらわしています。そして、dirmoveというメソッドが呼び出されるたびに、少しずつ移動していきます。turnというメソッドは、ボールが壁に当たったときに、その進行方向を変更します。また、ボールがラケットに当たったときは、hitというメソッドによって進行方向が変更されます。

参考文献

[RubyFAQ,2002] *Ruby FAQ*, maintained by Dave Thomas, 2002.
`http://www.rubycentral.com/faq/`

[Slagell,2002] Mark Slagell, *Teach Yourself Ruby in 21 Days*, Sams Publishing, 2002, ISBN 978-0-672-32252-5.

[Thomas,2005] David Thomas with Chad Fowler and Andrew Hunt, *Programming Ruby: The Pragmatic Programmer's Guide, Second Edition*, Pragmatic Programmers, 2005, ISBN 978-0-9745140-5-5. 第 I 部〜第 III 部の邦訳（田和勝）、『プログラミング Ruby・第 2 版・言語編』、オーム社、2006、ISBN 978-4-274-06642-9。第 IV 部の邦訳（田和勝）、『プログラミング Ruby・第 2 版・ライブラリ編』、オーム社、2006、ISBN 978-4-274-06643-6。

[青木,2002] 青木峰郎、『Ruby ソースコード完全解説』、インプレス、2002、ISBN 978-4-8443-1721-0。

[高橋,2002] 高橋征義、後藤裕蔵、『たのしい Ruby——Ruby ではじめる気軽なプログラミング——』、ソフトバンクパブリッシング、2002、ISBN 978-4-7973-1408-3。

[永井,2001] 永井秀利、『Ruby を 256 倍使うための本・界道編』、アスキー、2001、ISBN 978-4-7561-3993-1。

[原,2000] 原信一郎、『Ruby プログラミング入門』、オーム社、2000、ISBN 978-4-274-06385-5。

[前田,2002] 前田修吾、まつもとゆきひろ、やまだあきら、永井秀利、『Ruby アプリケーションプログラミング』、オーム社、2002、ISBN 978-4-274-06461-6。

[まつもと,1999] まつもとゆきひろ、石塚圭樹、『オブジェクト指向スクリプト言語 Ruby』、アスキー、1999、ISBN 978-4-7561-3254-3。

[まつもと,2000] まつもとゆきひろ、『Ruby デスクトップリファレンス』、オライリー・ジャパン、2000、ISBN 978-4-87311-023-3。

[まつもと,2003] まつもとゆきひろ、『Ruby 1.6 リファレンスマニュアル』、2003。
`http://www.ruby-lang.org/ja/man-1.6/`

[るびきち,2003] るびきち、『Ruby シェルプログラミング』、技術評論社、2003、ISBN 978-4-7741-1798-0。

索引