なでしこ実習マニュアル

第零版
2012 年 9 月 25 日（火）

目次

第 1 章　なでしこの基礎

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は、「文書」と呼ばれます。

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のような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」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」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」と呼ばれるものもあります。人間ではなくて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、通常、自然言語ではなくて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」と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラム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」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」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。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、……というように、枚挙にいとまがないほどです。

1.1.4　この文章について

この文章（「なでしこ実習マニュアル」）は、なでしこというプログラミング言語を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

1.1.5　なでしこについて

「なでしこ」というのは、クジラ飛行机（酒徳峰章）さんという人が設計したプログラミング言語です。

なでしこは、語彙や文法が日本語に近くなるように設計されています。ですから、日本人にとってとても親しみやすいプログラミング言語です。

なでしこを使うために必要なソフトは無料で配布されていて、なでしこの公式サイト[1]からダウンロードすることができます。

1.2　なでしこエディタ

1.2.1　なでしこエディタの基礎

なでしこのソフトをインストールすると、なでしこのフォルダが作られて、その中にさまざまなフォルダやファイルが作られます。

なでしこのフォルダの中に、nakopad.exe というファイルがあります。これは、「なでしこエディタ」と呼ばれるプログラムで、基本的にはテキストエディターの一種です。

[1] http://nadesi.com/

なでしこのプログラムは単なるテキストデータですので、それを書くためのテキストエディターは、どんなものでもかまいません。しかし、なでしこエディタは、テキストを編集することができるだけではなくて、入力されたテキストをなでしこのプログラムとして実行する機能も持っていますので、なでしこのプログラムを書く場合には、これを使うと便利です。

1.2.2 プログラムの入力

それでは、なでしこエディタを起動して、次のプログラムを入力してみてください。

プログラムの例 `hello.nako`

```
「こんにちは、世界。」と表示
```

ちなみに、これは、「こんにちは、世界。」という文字列を画面に表示する、という動作をするプログラムです。

1.2.3 プログラムの実行

入力されたプログラムは、右向きの三角形のボタンを押すか、メニューの「実行 → 実行」をクリックするか、またはファンクションキーの F5 を押すことによって実行することができます。

先ほど入力したプログラムを実行すると、「なでしこ」というウィンドウが開いて、そこに、「こんにちは、世界。」と表示されるはずです。

なでしこのプログラムを実行したときに開かれるウィンドウは、「母艦」と呼ばれます。なでしこのプログラムは、母艦を閉じることによって終了させることができます。

1.2.4 プログラムの保存

プログラムは、ファイルに保存することも可能です。プログラムをファイルに保存したいときは、フロッピーディスクのボタンを押すか、メニューの「ファイル → 保存」をクリックするか、またはコントロールキーを押しながら S のキーを押します。

なでしこのプログラムをファイルに保存する場合、ファイル名には、 `.nako` という拡張子を付けます。

それでは、先ほど入力したプログラムを、 `hello.nako` というファイル名のファイルに保存してみてください。

次に、なでしこのプログラムを保存したファイルのアイコンをダブルクリックしてみてください。そうすると、そのファイルに格納されているプログラムが起動するはずです。このように、ファイルに保存されているなでしこのプログラムは、そのファイルのアイコンをダブルクリックすることによって起動することができます。

1.2.5 プログラムの新規作成

なでしこエディタを使っていて、編集中のプログラムとは別の新しいプログラムの入力を始めたいときは、白紙のボタンを押すか、メニューの「ファイル → 新規」をクリックします。

1.2.6 エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」と呼ばれます。

エラーを含んでいるなでしこのプログラムを実行しようとすると、そのエラーについてのメッセージが表示されます。そのような、エラーについてのメッセージは、「エラーメッセージ」と呼ばれます。

それでは、エラーを含んでいるプログラムを実行してみましょう。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 `error.nako`

```
「こんにちは、世界。」が表示
```

これは、先ほどのプログラムの中の「と」を「が」に置き換えたものです。このプログラムを実行すると、「なでしこエラー表示」というウィンドウが開いて、そこにエラーメッセージが表示されます。

1.3　文

1.3.1　文の基礎

プログラムというのは、コンピュータに実行してほしい動作を記述した文書のことです。なでしこの場合、動作は、「文」と呼ばれるものを組み合わせることによって記述します。

個々の文は、コンピュータによる何らかの動作を意味しています。文が意味している動作をコンピュータが実行することを、「文を実行する」と言います。

第 1.2 節で紹介したプログラムは、

「こんにちは、世界。」と表示

というひとつの文から構成されています。また、この文の中に含まれている、

「こんにちは、世界。」

という部分も、ひとつの文です。このように、文というのは、その中に文を含んでいる（つまり入れ子になっている）場合があります。

2 個以上の文から構成されるプログラムを書くことも可能です。たとえば、次のプログラムは、3 個の文から構成されています。

プログラムの例　letsgo.nako

```
「じゅんで〜〜す！」と表示
「長作で〜〜す！」と表示
「三波春夫でございます」と表示
```

このプログラムを実行すると、

じゅんで〜〜す！
長作で〜〜す！
三波春夫でございます

と表示されます。この実行結果から、プログラムを構成しているそれぞれの文は、書かれている順番のとおりに実行される、ということが分かります。

1.3.2　文の値

文を実行すると、その結果として何らかのデータが得られます。文を実行したときに、その結果として得られたデータは、その文の「値」と呼ばれます。たとえば、

「こんにちは、世界。」

という文を実行すると、鍵括弧の内側にある文字列が結果として得られます。そして、その文字列がこの文の値になります。

文の中には、結果を求めることを目的としていないものもあります。そのような文の値は、「空文字列」と呼ばれる文字列になります。空文字列というのは、0 個の文字から構成される文字列のことです。たとえば、

「こんにちは、世界。」と表示

という文は、結果を求めることを目的とするものではありません。ですから、この文の値は空文字列になります。

1.3.3　命令

なでしこでは、動作を記述して、それに名前を付ける、ということができます。動作を記述して、それに名前を付けたものは、「命令」と呼ばれます。命令に与えられた名前は、「命令名」と呼ばれます。

命令を作ること、つまり、動作を記述して、それに名前を付けることは、命令を「定義する」と言われます。命令を定義するために書かれる記述は、「命令定義」と呼ばれます。

命令は、組み込み命令とユーザー定義命令に分類することができます。

「組み込み命令」というのは、プログラムの中に命令定義を書かなくても使うことのできる、あらかじめ定義されている命令のことです。なでしこは、さまざまな組み込み命令を持っています。たとえば、

「こんにちは、世界。」と表示

という文に含まれている「表示」という名前が示しているのは、母艦の上にデータを表示するという動作をする組み込み命令です。

「ユーザー定義命令」というのは、プログラムの中に定義を書くことによって定義された命令のことです。

命令は、「関数」と呼ばれることもあります。なでしこでは、「命令」と「関数」は、まったく同じ意味の言葉です。なお、このチュートリアルでは、「関数」という言葉は使わないで、「命令」という言葉で統一することにしたいと思います。

1.3.4 命令文

命令が意味している動作をコンピュータに実行させたいときは、「命令文」と呼ばれる、命令名などから構成される文を書きます。たとえば、

「こんにちは、世界。」と表示

という文は、「表示」という命令をコンピュータに実行させる命令文です。

1.3.5 引数

命令は、データを受け取って、そのデータを処理することができます。命令が受け取るデータは、「引数」と呼ばれます。たとえば、「表示」という命令は、表示する文字列を引数として受け取ります。

命令に対して引数を渡すためには、その命令を実行するための命令文の中に、その引数を書く必要があります。たとえば、「こんばんは。」という文字列を引数として「表示」に渡したいときは、

「こんばんは。」と表示

という命令文を書きます。この場合、引数と命令名とのあいだには、「と」または「を」という助詞を書く必要があります。助詞が必要かどうか、必要だとするとそれはどの助詞かというのは、それぞれの命令ごとに決まっています。

1.3.6 戻り値

命令は、データを返すこともできます。命令が返すデータは、「戻り値」と呼ばれます。

たとえば、「文字数」という命令は、戻り値を返します。

文字列を構成している文字の個数は、文字列の「文字数」と呼ばれます（「長さ」と呼ばれることもあります）。「文字数」という命令は、引数として文字列を受け取って（助詞は「の」）、その文字列の文字数を戻り値として返します。

命令が返した戻り値は、その命令を実行した命令文の値になります。たとえば、

「住まいが違います」の文字数

という命令文を実行すると、「文字数」は戻り値として 8 を返しますので、この命令文の値は 8 になります。

次のプログラムは、17 という数値を表示します。

プログラムの例 `mojisuu.nako`

```
「世界を崩したいなら泣いた雫を生かせ」の文字数を表示
```

1.4 改行と注釈

1.4.1 文と行の関係

なでしこの文は、ひとつの行の中に何個でも好きなだけ並べて書くことができます。たとえば、

「ワレラ」と表示「ロリー」と表示「コンダ」と表示

というように、ひとつの行の中に三つの文を書くことができます。

ひとつの行の中に何個かの文を書く場合、それらの文を句点（。）で区切ることもできます。たとえば、先ほどの三つの文は、

「ワレラ」と表示。「ロリー」と表示。「コンダ」と表示

というように書くこともできます。
　ひとつの文は、助詞の直後に読点（、）と改行を入れることによって、2 個以上の行に分けて書くことも可能です。たとえば、

```
「バルス」と表示
```

という文は、

```
「バルス」と、
表示
```

と書くこともできます。

1.4.2　注釈

　なでしこのプログラムの中には、文だけではなくて、「注釈」と呼ばれるものを書くこともできます。
　注釈というのは、プログラムが実行されるときに無視される文字列のことです。無視されるということは、注釈として書かれたものは、それがどんなものであろうともエラーにはならないということです。
　注釈には、いくつかの書き方があります。
　注釈の書き方のひとつは、シャープ（#）を使う書き方です。プログラムの中にシャープを書くと、そこから最初の改行までが注釈とみなされます。たとえば、

```
「のの子」と表示　#　第三小学校三年三組
「まつ子」と表示
```

と書いた場合、

```
#　第三小学校三年三組
```

という部分が注釈とみなされます。
　注釈のもうひとつの書き方として、注釈にしたい記述をスラッシュアスタリスク (/*) とアスタリスクスラッシュ(*/) で囲むという書き方もあります。たとえば、

```
/*ここは注釈です。
ここも注釈です。
そしてここも注釈です。*/
```

という記述は、注釈とみなされます。
　注釈を書く最大の目的は、それを書いた人間が、それを読む人間に対して、プログラムの意味を理解しやすくするためのヒントを与えることです。
　注釈の目的はそれだけではありません。プログラムの一部分を一時的に無効にしたい場合（つまりコンピュータに無視されるようにしたい場合）にも注釈が使われます。つまり、無効にしたい部分を注釈にするわけです。
　プログラムの一部分を注釈にすることによって無効にすることを、その部分を「コメントアウトする」と言います。

1.5　基本的な命令

1.5.1　データの表示

　この節では、基本的な命令をいくつか紹介したいと思います。
　まずは、データを表示するための命令です。
　「表示」は、引数（助詞は「と」または「を」）を母艦の上に表示して、そのあとに改行を表示します。ですから、この命令を何回か実行すると、表示されるデータは縦に並ぶことになります。
　データを表示したいけれども、そのあとの改行は表示したくない、という場合は、「継続表示」という命令を使います。この命令は、「表示」と同じように引数を表示しますが、そのあとに改行は表示しません。

プログラムの例　keizoku.nako

```
「菜の花や」と継続表示
「月は東に」と継続表示
「日は西に」と継続表示
```

1.5.2 ダイアログボックスによるデータの表示

「言う」は、「表示」と同じように、引数（助詞は「と」または「を」）を表示する命令です。ただし、「表示」とは違って母艦の上に表示するのではなくて、ダイアログボックスを開いて、その上に表示します。

プログラムの例 `iu.nako`

```
「私はダイアログボックスです。」と言う
```

1.5.3 文字列の読み込み

「尋ねる」は、文字列を読み込むためのダイアログボックスを開く命令です。引数（助詞は「で」または「と」または「を」）は、そのダイアログボックスの上に質問として表示されます。文字列を入力して、「決定」というボタンをクリックすると、ダイアログボックスが閉じて、入力された文字列が戻り値として返ってきます。「取消」というボタンをクリックした場合、戻り値として空文字列が返ってきます。

プログラムの例 `tazuneru.nako`

```
「文字列を入力してください。」と尋ねるを表示
```

1.5.4 加減乗除

なでしこには、次のような加減乗除の命令があります。

足す	加算。A に B を足す
引く	減算。A から B を引く
掛ける	乗算。A に B を掛ける
割る	除算。A を B で割る
割った余り	剰余。A を B で割った余り
乗	べき乗。A の B 乗

たとえば、

```
5に3を足す
```

という文を実行すると、5と3が引数として「足す」に渡されます。「足す」は、受け取った引数を加算して、その結果を戻り値として返します。ですから、この文の値は8になります。

プログラムの例 `nijou.nako`

```
「数値を入力してください。」と尋ねるの2乗を表示
```

1.5.5 丸括弧

先ほど、2乗を求めるプログラムを紹介しましたが、2のべき乗を求めるプログラムは、どのように書けばいいのでしょうか。

先ほどのプログラムの中にある文の語順を入れ替えた、

```
2の「数値を入力してください。」と尋ねる乗を表示
```

という文を実行すると、エラーメッセージが表示されます。これは、「尋ねる」という命令名と「乗」という命令名とがつながって、「尋ねる乗」という存在しない命令の名前になってしまったからです。

この文が正しく解釈されるようにするためには、丸括弧を使う必要があります。丸括弧で文を囲むと、その全体がひとつの文になります。そして、丸括弧で囲まれている文は、どのような文の中に書かれていても、それ自体で文として完結していると解釈されます。ですから、先ほどの文は、

```
2の（「数値を入力してください。」と尋ねる）乗を表示
```

と書くことによって、正しく解釈されるようになります。

プログラムの例 `ninobeki.nako`

```
2 の（「数値を入力してください。」と尋ねる）乗を表示
```

1.5.6　切り下げ

数値を扱うプログラムでは、しばしば、小数を整数にすることが必要になります。

小数を整数に変換したいときに使われるのは、「切り下げ」という命令です。

切り下げ は、引数として数値を受け取って（助詞は「を」）、その数値を超えない最大の整数を求めて、その整数を戻り値として返します。たとえば、5.3 を「切り下げ」に渡した場合の戻り値は 5 になって、−5.3 を「切り下げ」に渡した場合の戻り値は −6 になります。

プログラムの例　`kirisage.nako`

```
5.3 を切り下げを表示
-5.3 を切り下げを表示
```

1.6　リテラル

1.6.1　リテラルの基礎

特定のデータをあらわしている文は、「リテラル」と呼ばれます。たとえば、2 という文は、2 という特定のデータをあらわしていますので、リテラルに分類されます。

リテラルを実行すると、それがあらわしているデータが、その値として得られます。たとえば、2 というリテラルを実行すると、それがあらわしている 2 というデータが、その値として得られます。

1.6.2　数値リテラル

数値をあらわしているリテラルは、「数値リテラル」と呼ばれます。たとえば、2 というリテラルは、数値をあらわしていますので、数値リテラルに分類されます。

プラスの整数をあらわす数値リテラルは、0 から 9 までの数字を並べることによって作ります。たとえば、6081 は、6081 というプラスの整数をあらわしている数値リテラルです。

小数をあらわす数値リテラルは、小数点を示すドット (.) を書くことによって作ることができます。たとえば、3.14 は、3.14 という小数をあらわしている数値リテラルです。

プラスの数をあらわす数値リテラルの左側にマイナス (-) を書いたものは、マイナスの数値をあらわす文になります。たとえば、-6081 はマイナスの 6081 をあらわしている文です。

1.6.3　文字列リテラル

文字列をあらわしているリテラルは、「文字列リテラル」と呼ばれます。

文字列リテラルは、鍵括弧で文字列を囲むことによって作られます。たとえば、

「こんにちは、世界。」

というのは、「こんにちは、世界。」という文字列をあらわしている文字列リテラルです。

文字列リテラルは、改行を含んでいてもかまいません。文字列リテラルの中に含まれている改行は、それがあらわしている文字列の中にも含まれることになります。

プログラムの例　`kaigyou.nako`

```
「私は、
改行を含んでいる
文字列です。」と表示
```

1.6.4　文展開

中括弧（{}）で文を囲んだ文字列は、「文展開」と呼ばれます。

文展開を文字列のリテラルの中に書くと、そのリテラルが実行された時点で、文展開の中の文も実行されて、その値が文字列の中に埋め込まれます。たとえば、

「7 に 8 を掛けた結果は {7 に 8 を掛ける} です。」

というリテラルを実行すると、その中に書かれた文がその時点で実行されて、

「7 に 8 を掛けた結果は 56 です。」

という文字列が生成されます。

プログラムの例 buntenkai.nako

「7 に 8 を掛けた結果は｛7 に 8 を掛ける｝です。」と表示

1.7 変数

1.7.1 変数の基礎

なでしこのプログラムでは、データを箱を入れておくということができます。データを入れておくための箱は、「変数」と呼ばれます。

変数という箱にデータを入れることは、変数にデータを「代入する」と言われます。

変数は、それに与えられた名前によって識別されます。変数に与えることのできる名前は、「識別子」と呼ばれます。

1.7.2 識別子の作り方

識別子は、次のような規則に従って作ることになっています。

- 識別子を作るために使うことのできる文字は、漢字、ひらがな、カタカナ、英字、数字です。
- 識別子の先頭の文字として、数字を使うことはできません。
- 予約語と同じものは識別子としては使えません。「予約語」というのは、用途があらかじめ予約されている単語のことで、「もし」「違えば」「繰り返す」「ループ」「間」などがあります。
- 「と」「を」「の」「に」「へ」「で」「は」「から」「まで」などの助詞が末尾にある単語は、識別子としては使えません。

識別子として使うことのできるものの例としては、次のようなものがあります。

甲　半径　まろやかさ　ボリューム　happiness　嘘 800

他方、次のようなものを識別子として使うことはできません。

人間@場所　使うことのできない文字を含んでいる。

5 大老　先頭の文字が数字。

もし　同じ予約語が存在する。

こだから　「から」という助詞が末尾にある。

1.7.3 変数の宣言

変数を作ることを、変数を「宣言する」と言います。

変数を宣言するという動作は、「変数宣言」と呼ばれるものによって記述されます。

変数宣言は、基本的には、

変数名 とは 型名

と書きます。「変数名」のところには、変数に名前として与えたい識別子を書きます。そして、「型名」のところには、変数に入れるデータの種類をあらわす単語を書きます。たとえば、変数に入れるデータが整数ならば「整数」、整数とは限らない数値ならば「数値」、文字列ならば「文字列」と書きます。

変数宣言は、変数を宣言して、その名前として識別子を与えます。たとえば、

氏名とは文字列

という変数宣言は、文字列を入れる変数を作って、それに対して「氏名」という名前を与えます。

変数宣言は、変数を宣言するという動作だけではなくて、宣言された変数を初期化するという動作も記述することができます。「初期化する」というのは、宣言と同時に変数にデータを代入するということです。宣言と同時に変数に代入されるデータは、「初期値」と呼ばれます。

変数を宣言して初期化したいときは、

変数名 とは 型名 ＝ 文

という変数宣言を書きます。この形の変数宣言は、イコール（＝）の右側に書かれた文の値を、変数に対して初期値として代入します。たとえば、

半径とは数値　＝　7

という変数宣言を書くことによって、数値を入れる変数を作って、それに対して「半径」という名前を与えて、その変数に 7 という初期値を代入します。

変数宣言の中のイコールの前と後ろには、上の例のように 1 個の空白を書くのが普通ですが、それらの空白は絶対に必要というわけではありませんので、

半径とは数値＝7

というように詰めて書いても問題はありません。

1.7.4　識別子の値

変数に名前として与えられている識別子は、文として実行することができます。識別子を実行すると、その識別子が与えられている変数に代入されているデータが、その値として得られます。

プログラムの例　hensuu.nako

```
甲とは数値　＝　「1 個目の数値を入力してください。」と尋ねる
乙とは数値　＝　「2 個目の数値を入力してください。」と尋ねる
「{甲} に {乙} を足すと {甲に乙を足す}」と表示
「{甲} から {乙} を引くと {甲から乙を引く}」と表示
「{甲} に {乙} を掛けると {甲に乙を掛ける}」と表示
「{甲} を {乙} で割ると {甲を乙で割る}」と表示
「{甲} を {乙} で割った余りは {甲を乙で割った余り}」と表示
「{甲} の {乙} 乗は {甲の（乙）乗}」と表示
```

1.7.5　それ

なでしこでは、「それ」という名前の特殊な変数が、あらかじめ宣言されています。

第 1.3.6 項で説明したように、戻り値を返す命令を実行したとき、その戻り値は、その命令を実行した命令文の値になるわけですが、それだけではなくて、その戻り値は「それ」にも代入されます。

ですから、命令の戻り値を利用する方法には二通りのものがある、ということになります。ひとつは、文の中に文を書くことによって、外側の文が内側の文の値を利用する、という方法で、もうひとつは、「それ」という変数の内容を利用するという方法です。

プログラムの例　sore.nako

```
「数値を入力してください。」と尋ねる
「{それ} を 3 倍した結果は {それに 3 を掛ける} です。」と表示
```

第 2 章　選択

2.1　選択の基礎

2.1.1　選択とは何か

いくつかの文を並べて書くことによって、まずこの動作を実行して、次にこの動作を実行して、次にこの動作を実行して……というように、複数の動作を直線的に実行していくという動作を記述することができるわけですが、コンピュータに実行させたい動作は、必ずしも一直線に進んでいくものばかりとは限りません。しばしば、そのときの状況に応じて、いくつかの動作の候補の中からひとつの動作を選んで実行するということも、必要になります。

「いくつかの動作の候補の中からひとつの動作を選んで実行する」という動作は、「選択」と呼ばれます。

2.1.2　条件

コンピュータは、運を天に任せて動作を選択するわけではありません。選択は、何らかの判断にもとづいて実行されます。

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」であると言われます。

2.1.3 真偽値

真を意味するデータと、偽を意味するデータは、総称して「真偽値」と呼ばれます。

なでしこでは、真は 1 という整数によってあらわされ、偽は 0 という整数によってあらわされます。

2.2 比較命令

2.2.1 比較命令の基礎

二つのデータを比較して、それらのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる命令は、「比較命令」と呼ばれます。

比較命令は、二つのデータを引数として受け取ります。そして、それらのデータのあいだに関係が成り立っているかどうかという判断を実行して、関係が成り立っているならば 1、成り立っていないならば 0 を戻り値として返します。

2.2.2 大小関係

次の比較命令を使うことによって、数値の大小関係について調べることができます。

超　　A が B 超。たとえば「5 が 3 超」は 1、「5 が 8 超」は 0、「5 が 5 超」も 0。

未満　A が B 未満。たとえば「5 が 8 未満」は 1、「5 が 3 未満」は 0、「5 が 5 未満」も 0。

以上　A が B 以上。たとえば「5 が 3 以上」は 1、「5 が 8 以上」は 0、「5 が 5 以上」は 1。

以下　A が B 以下。たとえば「5 が 8 以下」は 1、「5 が 3 以下」は 0、「5 が 5 以下」は 1。

プログラムの例　daishou.nako

```
甲とは数値　＝　「1 個目の数値を入力してください。」と尋ねる
乙とは数値　＝　「2 個目の数値を入力してください。」と尋ねる
「{甲} が {乙} 超の値は {甲が（乙）超}」と表示
「{甲} が {乙} 未満の値は {甲が（乙）未満}」と表示
「{甲} が {乙} 以上の値は {甲が（乙）以上}」と表示
「{甲} が {乙} 以下の値は {甲が（乙）以下}」と表示
```

2.2.3 等しいかどうか

二つのデータが等しいかどうかということは、次の比較命令を使うことによって調べることができます。

等しい　A が B と等しい。たとえば「5 が 5 と等しい」は 1、「5 が 3 と等しい」は 0。

ない　　A が B でない。たとえば「5 が 3 でない」は 1、「5 が 5 でない」は 0。

プログラムの例　hitoshii.nako

```
甲とは文字列　＝　「1 個目の文字列を入力してください。」と尋ねる
乙とは文字列　＝　「2 個目の文字列を入力してください。」と尋ねる
「{甲} が {乙} と等しいの値は {甲が乙と等しい}」と表示
「{甲} が {乙} でないの値は {甲が乙でない}」と表示
```

2.3 もし文

2.3.1 もし文の基礎

何らかの条件が成り立っているかどうかを調べて、その結果にもとづいて二つの動作のうちのどちらかを実行したい、というときは、「もし文」という文を使います。

もし文は、

もし 条件文 ならば

　プログラム $_1$

違えば
　プログラム $_2$

と書きます。「条件文」のところには、条件をあらわす文、つまり、実行すると値として真偽値が得られる文を書きます。

もし文を実行すると、まず最初に、その中の条件文が実行されます。そして、条件文の値が真だった場合は、プログラム $_1$ が実行されます（その場合、プログラム $_2$ は実行されません）。条件文の値が偽だった場合は、プログラム $_2$ が実行されます（その場合、プログラム $_1$ は実行されません）。

プログラムの例　guusuu.nako

```
甲とは整数　＝　「整数を入力してください。」と尋ねる
「{甲} は」と継続表示
もし甲を 2 で割った余りが 0 と等しいならば
　「偶数」と継続表示
違えば
　「奇数」と継続表示
「です。」と表示
```

行の先頭に空白を書くことによって、プログラムの一部分を右にずらして書くことを、その部分を「インデントする」と言います。

もし文の中に書かれるプログラム $_1$ とプログラム $_2$ は、かならずインデントする必要があります。

2.3.2　「違えば」以降を省略したもし文

二つの動作のうちのどちらかを選択するのではなくて、ひとつの動作を実行するかしないかを選択したい、ということもしばしばあります。そのような場合は、「違えば」以降を省略したもし文を書きます。つまり、

もし 条件文 ならば
　プログラム

という形のもし文を書くわけです。この形のもし文を実行すると、条件文の値が真の場合はプログラムが実行されますが、条件文の値が偽の場合は何も実行されません。

プログラムの例　jikanfun.nako

```
分とは整数　＝　「時間の長さを分で入力してください。」と尋ねる
「{分} 分は」と継続表示
「{分を 60 で割るを切り下げ} 時間」と継続表示
もし分を 60 で割った余りが 0 でないならば
　「{分を 60 で割った余り} 分」と継続表示
「です。」と表示
```

これは、何分という形式であらわされた時間の長さを、何時間何分という形式に変換するプログラムです。ただし、変換すると何分の部分が 0 分になる場合は、何分の部分を省略します。何分の部分を出力するという動作は、実行するかしないかを選択することになりますので、「違えば」以降を省略したもし文を使って書かれています。

2.3.3　多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」と呼ばれます。多肢選択には、次の二つのタイプがあります。

- ひとつの文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。
- 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。

ひとつの文の値による多肢選択は、「条件分岐文」という文を使うことによって実行することができます（「条件分岐文」については次の節で説明します）。

いくつかの条件による多肢選択は、もし文を使うことによって記述することができます。その方法については、次の項で説明します。

2.3.4 違えばもし

もし文を使うことによって、いくつかの条件による多肢選択を記述したいときは、「もし…ならば…違えばもし…ならば…」というように、条件と同じ個数のもし文を、「違えば」の後ろにさらにもし文を書くという形で連結していきます。

たとえば、3 個の条件による多肢選択は、

もし 条件文 $_1$ ならば
　プログラム $_1$
違えばもし 条件文 $_2$ ならば
　プログラム $_2$
違えばもし 条件文 $_3$ ならば
　プログラム $_3$
違えば
　プログラム $_4$

というように、3 個のもし文を連結することによって記述することができます。この形のもし文を実行すると、値が真になる条件文が見つかるまで、条件文 $_1$、条件文 $_2$、条件文 $_3$ という順番で条件文が実行されていきます。値が真になる条件文が見つかった場合は、その条件文と同じ番号のプログラムが実行されます（その場合、それ以降の条件文は実行されません）。すべての条件文の値が偽だった場合は、プログラム $_4$ が実行されます。

プログラムの例　fugou.nako

```
甲とは数値　＝　「数値を入力してください。」と尋ねる
「{甲} は」と継続表示
もし甲が 0 超ならば
　「プラス」と継続表示
違えばもし甲が 0 未満ならば
　「マイナス」と継続表示
違えば
　「ゼロ」と継続表示
「です。」と表示
```

2.4 条件分岐文

2.4.1 条件分岐文の基礎

前の節で説明したように、ひとつの文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多肢選択は、「条件分岐文」という文を使うことによって記述することができます。

条件分岐文は、

判断文 で条件分岐
　選択肢の列

と書きます。「判断文」のところには、動作を選択するための値を求める文を書きます。

条件分岐文を実行すると、まず最初に、その中の判断文が実行されます。そして、その値が何なのかということによって、選択肢の中からひとつが選択されて、それが実行されます。

条件分岐文の中には、判断文の値によって選択されるいくつかの選択肢を書く必要があります。選択肢は、インデントが必要です。選択肢として書くことができるものとしては、次の 2 種類のものがあります。

- ならば節
- 違えば節

2.4.2 ならば節

ならば節は、

文ならば
　プログラム

と書きます。ならば節の先頭に書かれる、「文ならば」という部分は、「ならばラベル」と呼ばれます。ならば節の中のプログラムは、インデントが必要です。

ならば節で書かれた選択肢は、判断文の値とならばラベルの中に書かれた文の値とが一致した場合に選択されて、その中のプログラムが実行されます。

プログラムの例　menu.nako

```
「1 焼きそば
2 お好み焼き
3 たこ焼き
注文したいものの番号を入力してください。」と尋ねるで条件分岐
  1ならば
    「あなたは焼きそばを注文しました。」と表示
  2ならば
    「あなたはお好み焼きを注文しました。」と表示
  3ならば
    「あなたはたこ焼きを注文しました。」と表示
```

2.4.3　違えば節

違えば節は、

違えば
　プログラム

と書きます。違えば節の中のプログラムは、インデントが必要です。

違えば節を条件分岐文の最後の選択肢として書いておくと、その選択肢は、ならば節で書かれたどの選択肢も選択されなかった場合（つまり、ならばラベルの中に書かれたどの文の値も、判断文の値と一致しなかった場合）に選択されて、その中のプログラムが実行されます。

プログラムの例　chigaeba.nako

```
「1 コーヒー
2 紅茶
3 ジュース
注文したいものの番号を入力してください。」と尋ねるで条件分岐
  1ならば
    「あなたはコーヒーを注文しました。」と表示
  2ならば
    「あなたは紅茶を注文しました。」と表示
  3ならば
    「あなたはジュースを注文しました。」と表示
  違えば
    「その番号は無効です。」と表示
```

2.5　論理命令

2.5.1　論理命令の基礎

処理の対象が真偽値で、処理の結果も真偽値であるような動作をあらわしている命令は、「論理命令」と呼ばれます。

なでしこには、「かつ」や「または」や「ＮＯＴ」などの論理命令があります。

2.5.2　論理積命令

「かつ」を実行する文は、

文かつ文

と書きます。

「かつ」は、「論理積命令」と呼ばれます。これは、二つの条件が両方とも成り立っているかどうかを判断したいときに使われる論理命令で、次のような動作をします。

真　かつ　真　　⟶　真
真　かつ　偽　　⟶　偽
偽　かつ　真　　⟶　偽
偽　かつ　偽　　⟶　偽

2.5.3　論理和命令

「または」を実行する文は、

文 または 文

と書きます。

「または」は、「論理和命令」と呼ばれます。これは、二つの条件のうちの少なくとも一つが成り立っているかどうかを判断したいときに使われる論理命令で、次のような動作をします。

真　または　真　　⟶　真
真　または　偽　　⟶　真
偽　または　真　　⟶　真
偽　または　偽　　⟶　偽

プログラムの例　uruudoshi.nako

```
年とは整数　＝　「年を西暦で入力してください。」と尋ねる
「{年} 年はうるう年」と継続表示
条件 1 とは整数　＝　年を 4 で割った余りが 0 と等しい
条件 2 とは整数　＝　年を 100 で割った余りが 0 でない
条件 3 とは整数　＝　年を 400 で割った余りが 0 と等しい
もし（(条件 1）かつ（条件 2)）または（条件 3）ならば
　「です。」と表示
違えば
　「ではありません。」と表示
```

2.5.4　論理否定命令

「ＮＯＴ」を実行する文は、

ＮＯＴ（文）

と書きます。

「ＮＯＴ」は、「論理否定命令」と呼ばれます。これは、真偽値を反転させたいとき、つまり、「*A* である」という条件から「*A* ではない」という条件を作りたいときに使われる論理命令で、次のような動作をします。

ＮＯＴ（真）　　⟶　偽
ＮＯＴ（偽）　　⟶　真

第３章　繰り返し

3.1　繰り返しの基礎

3.1.1　繰り返しとは何か

コンピュータに実行させたい動作は、必ずしも、一連の動作をそれぞれ一回ずつ実行していけばそれで達成される、というものばかりとは限りません。しばしば、ほとんど同じ動作を何回も何十回も何百回も実行しなければ意図していることを達成できない、ということがあります。

「同じ動作を何回も実行する」という動作は、「繰り返し」と呼ばれます。

この章では、繰り返しというのはどのように記述すればいいのか、ということについて説明します。

3.1.2　繰り返しを記述するための文

繰り返しは、繰り返したい回数と同じ個数だけ、繰り返したいことを書くことによって記述することも可能ですが、そのような書き方だと、繰り返しの回数に比例してプログラムが長くなってしまいます。ですから、多くのプログラミング言語は、繰り返しを簡潔に記述することができるようにする機能を持っています。

なでしこでは、「回文」「繰り返す文」「間文」「反復文」などの文を使うことによって、繰り返しを簡潔に記述することができます。

それらの文のうちで、この章で紹介するのは、「回文」「繰り返す文」「間文」の三つです。「反復文」については、第 7.3 節で説明することにしたいと思います。

3.2　回文

3.2.1　回文の基礎

特定の回数だけ何らかの動作を繰り返したいときは、「回文」という文を使います。

回文は、

```
回数文 回
  プログラム
```

と書きます。「回数文」のところには、繰り返しの回数を求める文を書きます。回文の中のプログラムは、インデントが必要です。

回文は、回数文の値として得られた回数だけプログラムの実行を繰り返すという動作をあらわしています。

プログラムの例　kai.nako

```
甲とは文字列　＝　「文字列を入力してください。」と尋ねる
乙とは整数　＝　「回数を入力してください。」と尋ねる
乙回
  甲を言う
```

3.2.2　回数

回文の中では、「回数」という変数を使うことができます。この変数には、繰り返しが何回目なのかということを示す整数が代入されます。「回数」は、なでしこ自身によってあらかじめ宣言されている変数ですので、プログラムの中で宣言する必要はありません。

プログラムの例　kaisuu.nako

```
甲とは整数　＝　「回数を入力してください。」と尋ねる
甲回
  「{回数} 回目の実行です。」と言う
```

3.3　繰り返す文

3.3.1　繰り返す文の基礎

変数の内容を 1 ずつ変化させながら何らかの動作を繰り返したいときは、「繰り返す文」という文を使います。繰り返す文で使われる、内容が 1 ずつ変化する変数は、「制御変数」と呼ばれます。

繰り返す文は、

```
制御変数名 で 開始文 から 終了文 まで繰り返す
  プログラム
```

と書きます。「制御変数名」のところには、制御変数として使う変数の名前を書きます。そして、

「開始文」のところには制御変数に最初に代入する整数を求める文を書いて、「終了文」のところには、制御変数に最後に代入する整数を求める文を書きます。繰り返す文の中のプログラムは、インデントが必要です。

繰り返す文は、制御変数の内容を変化させながら、その中のプログラムの実行を繰り返す、という動作をあらわしています。制御変数の内容は、開始文の値から終了文の値まで、1 ずつ変化していきます。たとえば、

```
甲で 3 から 7 まで繰り返す
  甲を言う
```

という繰り返す文は、「甲」という変数の内容を 3、4、5、6、7 と変化させて、「甲を言う」を繰り返します。同じように、

```
甲で 7 から 3 まで繰り返す
  甲を言う
```

という繰り返す文は、「甲」という変数の内容を 7、6、5、4、3 と変化させて、「甲を言う」を繰り返します。

プログラムの例 kurikaesu.nako

```
甲とは整数 ＝ 「開始する整数を入力してください。」と尋ねる
乙とは整数 ＝ 「終了する整数を入力してください。」と尋ねる
丙とは整数
丙で甲から乙まで繰り返す
  「制御変数の内容は {丙} です。」と言う
```

3.3.2 繰り返す文の中の回数

回文の場合と同じように、繰り返す文の中でも、「回数」という変数を使うことができます。繰り返す文の場合も、この変数には、繰り返しが何回目なのかということを示す整数が代入されます。

プログラムの例 kaisuu2.nako

```
甲とは整数 ＝ 「開始する整数を入力してください。」と尋ねる
乙とは整数 ＝ 「終了する整数を入力してください。」と尋ねる
丙とは整数
丙で甲から乙まで繰り返す
  「制御変数の内容は {丙} で、回数は {回数} です。」と言う
```

3.4 代入文

3.4.1 代入文の基礎

第 1.7 節で説明したように、変数というのはデータを入れることのできる箱のことで、変数にデータを入れることを、変数にデータを「代入する」と言います。

変数にデータを代入したいときは、「代入文」という文を使います。

代入文は、

```
変数名 は 文
```

と書きます。「変数名」のところには、データを代入したい変数の名前を書きます。代入文を実行すると、その中に書かれた文が実行されて、その値が変数に代入されます。たとえば、

```
甲は 567
```

という代入文を実行すると、「甲」という変数に 567 が代入されます。

ひとつの変数には、ひとつのデータしか入れることができません。ですから、変数にデータを代入すると、それ以前の変数の内容は消えてしまいます。

3.4.2 処理の結果をもとの変数に戻す代入文

代入文は、変数の内容を処理して、その結果をもとの変数に代入する、という動作を記述することもできます。たとえば、

```
甲は 20
甲は甲に 3 を足す
```

という二つの代入文を実行したとしましょう。ひとつ目の代入文を実行すると、甲の内容は 20 になります。二つ目の代入文は、甲という変数から取り出したデータを処理して、その結果をもとの変数に戻しています。この代入文を実行すると、20 と 3 とを足し算した結果が甲に代入されますので、甲の内容は 23 になります。

3.4.3　変化の蓄積

繰り返しの中で変数に代入をすることによって、何らかの変化を蓄積していく、ということができます。

変化の蓄積の例として、階乗を求めるプログラムについて考えてみましょう。

n が 0 またはプラスの整数だとするとき、n の「階乗」というのは、1 から n までの範囲にあるすべての整数を掛け算した結果のことです（0 の階乗は 1 だと定義します）。つまり、

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

という計算をすることによって、n の階乗を求めることができるということです。

この計算をコンピュータに実行させたいときは、変数を使って変化を蓄積していきます。そのための変数として、「階乗」という変数を使うことにしましょう。

「階乗」は、1 で初期化しておきます。そして、2 から n までの範囲にあるそれぞれの整数について、その整数と「階乗」とを掛け算して、その結果を「階乗」に戻す、という処理を繰り返します。そうすると、繰り返しが終わったとき、「階乗」には n の階乗が入っているはずです。

プログラムの例　kaijou.nako

```
甲とは整数　＝　「整数を入力してください。」と尋ねる
階乗とは整数　＝　1
もし甲が 2 以上ならば
  乙とは整数
  乙で甲から 2 まで繰り返す
    階乗は階乗に乙を掛ける
違えば
  階乗は 1
「{甲} の階乗は {階乗} です。」と表示
```

3.5　間文

3.5.1　間文の基礎

何らかの条件が成り立っているあいだだけ、何らかの動作を繰り返したい、というときは、「間文」という文を使います。

間文は、

条件文 の間

　プログラム

と書きます。「条件文」のところには、条件をあらわす文、つまり、実行すると値として真偽値が得られる文を書きます。間文の中のプログラムは、インデントが必要です。

間文は、次のような動作をあらわしています。

(1) 条件文を実行する。その値が偽だった場合、間文の動作は終了する。

(2) 条件文の値が真だった場合は、プログラムを実行する。

(3) (1) に戻って、ふたたび同じ動作を実行する。

回文や繰り返す文の場合と同じように、間文の中でも、「回数」という変数を使うことができます。間文の場合も、この変数には、繰り返しが何回目なのかということを示す整数が代入されます。

3.5.2 最大公約数

間文を使うことによって素直に記述することができる繰り返しの例として、二つの整数の最大公約数を求めるという処理について考えてみましょう。

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、54 と 36 の最大公約数は 18 です。

二つの整数の最大公約数は、「ユークリッドの互除法」と呼ばれる方法を使えば、とても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、甲と乙という変数に代入する。

ステップ 2　乙が 0 ならば計算を終了する。

ステップ 3　甲を乙で除算して、その余りを丙という変数に代入する。

ステップ 4　乙を甲に代入する。

ステップ 5　丙を乙に代入する。

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、最初に与えられた二つの整数の最大公約数が甲の中に入っている、というものです。ステップ 2 からステップ 6 までは、

　乙が 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、間文を書くことによって素直に記述することができます。

プログラムの例　gcm.nako

```
甲とは整数　＝　「1 個目の整数を入力してください。」と尋ねる
乙とは整数　＝　「2 個目の整数を入力してください。」と尋ねる
「{甲} と {乙} の最大公約数は」と継続表示
丙とは整数
乙が 0 でないの間
  丙は甲を乙で割った余り
  甲は乙
  乙は丙
「{甲} です。」と表示
```

第 4 章　命令の定義

4.1　命令の定義の基礎

4.1.1 命令についての復習

この章では、命令を定義する方法について説明したいと思います。そこで、それに先立って、命令について第 1.3 節で説明したことを復習しておくことにしましょう。

命令というのは、動作を記述して、それに名前を付けたもののことです。命令に与えられた名前は、「命令名」と呼ばれます。

命令は、なでしこによって最初から定義されているものと、プログラムの中で定義されたものに分類することができます。なでしこによって最初から定義されている命令は「組み込み命令」と呼ばれ、プログラムの中で定義された命令は「ユーザー定義命令」と呼ばれます。

命令は、データを受け取って、そのデータを処理して、その結果を返すことができます。命令が受け取るデータは「引数」と呼ばれ、命令が返すデータは「戻り値」と呼ばれます。

命令を実行したいときは、「命令文」と呼ばれる文をプログラムの中に書きます。命令が返した戻り値は、その命令を実行した命令文の値になります。

4.1.2 命令定義

命令を定義したいときは、「命令定義」と呼ばれる記述をプログラムの中に書きます。

引数を受け取らない命令を定義する命令定義は、

●命令名

プログラム

と書きます。「命令名」のところには、定義される命令に名前として与えたい識別子を書きます。命令定義の中のプログラムは、インデントが必要です。

たとえば、「こんにちは。」という文字列を母艦に表示する、「あいさつする」という命令名を持つ命令を定義したいときは、

```
●あいさつする
  「こんにちは。」と表示
```

という命令定義を書きます。

命令定義と、それによって定義された命令を実行する命令文との位置関係は、どちらが上でどちらが下でもかまいません。

プログラムの例　meireiteigi.nako

```
あいさつする

●あいさつする
  「こんにちは。」と表示
```

4.2　スコープ

4.2.1　スコープの基礎

識別子と、その識別子が名前として与えられている対象とのあいだの関係が保たれる、プログラムの上での範囲は、その識別子の「スコープ」と呼ばれます。

プログラミング言語の多くは、識別子のスコープに関する規則を定めています。なでしこも例外ではありません。

4.2.2　グローバルなスコープ

プログラムの全域というスコープは、「グローバルなスコープ」と呼ばれます。グローバルなスコープを持つ識別子が与えられた変数は、「グローバル変数」と呼ばれます。

なでしこでは、命令定義の外で変数に与えられた識別子は、グローバルなスコープを持つことになります。

命令定義の内部も、グローバルなスコープの一部分になります。ですから、命令定義の中でも、グローバル変数に与えられた識別子を書くことによって、その変数の内容を求めたり、その変数にデータを代入したりすることができます。

プログラムの例　global.nako

```
甲とは文字列　＝　「私はグローバル変数です。」
甲を表示
命令
甲を表示

●命令
  甲を表示
  甲は「命令定義の中で違うデータを代入してみました。」
  甲を表示
```

4.2.3　ローカルなスコープ

プログラムの全域よりも狭い、限定されたスコープは、「ローカルなスコープ」と呼ばれます。ローカルなスコープを持つ識別子が与えられた変数は、「ローカル変数」と呼ばれます。

グローバルなスコープを持つ識別子と同一の識別子を、ローカルなスコープを持つ識別子として使っても、問題はありません。ただし、その場合、ローカルなスコープの内部では、グローバルなスコープを持つ識別子を名前として持つものは、隠されて見えなくなります。

なでしこでは、命令定義の中で変数に与えられた識別子は、その命令定義の中だけというスコープを持つことになります。

プログラムの例 local.nako

```
甲とは文字列 ＝ 「私はグローバル変数の甲です。」
甲を表示
命令一
甲を表示

●命令一
  甲とは文字列 ＝ 「私は命令一のローカル変数の甲です。」
  甲を表示
  命令二
  甲を表示

●命令二
  甲とは文字列 ＝ 「私は命令二のローカル変数の甲です。」
  甲を表示
```

4.2.4 ローカルなスコープという規則のメリット

ところで、「命令定義の中で変数に与えられた識別子は、その命令定義の中だけというスコープを持つ」という規則には、いったいどのようなメリットがあるのでしょうか。

もしも、「命令定義の中で変数に与えられた識別子もグローバルなスコープを持つ」という規則が定められていたとするとどうなるか、ということについて考えてみましょう。その場合、変数に識別子を与えるときには、その識別子がすでに別の命令定義の中で使われていないか、ということに細心の注意を払う必要があります。うっかりと同一の識別子を複数の命令定義の中で使うと、思わぬ不具合が発生しかねません。

つまり、「命令定義の中で変数に与えられた識別子は、その命令定義の中だけというスコープを持つ」という規則は、「命令定義を書くときに、その命令定義の外でどのような識別子が使われているかということを、まったく気にする必要がない」というメリットを、プログラムを書く人に与えてくれているのです。

4.3 引数を受け取る命令の定義

4.3.1 仮引数

命令が受け取った変数は、「仮引数」と呼ばれる変数に格納されます。ですから、引数を受け取る命令を定義するためには、その引数を受け取る仮引数を、受け取る引数と同じ個数だけ宣言する必要があります。

引数を受け取る命令を定義したいときは、

●命令名（仮引数宣言）
プログラム

という形の命令定義を書きます。つまり、命令名の右側に丸括弧を書いて、その丸括弧の中に、「仮引数宣言」と呼ばれるものを書くわけです。

仮引数宣言は、その名前のとおり、仮引数を宣言する記述で、

仮引数名 助詞

という形のものを、必要な仮引数の個数だけ並べたものです。「仮引数名」のところには、名前として仮引数に与える識別子を書いて、「助詞」のところには、引数を求める文の直後に書く助詞を書きます。

プログラムの例 karihikisuu.nako

```
「誰にあいさつしますか。」と尋ねるにあいさつする

●あいさつする（相手に）
  「{相手}さん、こんにちは。」と表示
```

なでしこでは、仮引数に与えられた識別子は、命令定義の中だけというスコープを持つことになります。つまり、仮引数というのはローカル変数だということです。

4.3.2　助詞

なでしこで助詞として使うことのできる単語は、次のとおりです。

から、が、くらい、して、だけ、って、て、で、でなければ、では、と、として、とは、など、なのか、なら、ならば、に、について、の、は、へ、ほど、まで、までの、までを、より、を

プログラムの例　joshi.nako

```
甲とは整数　＝　「開始する整数を入力してください。」と尋ねる
乙とは整数　＝　「終了する整数を入力してください。」と尋ねる
甲から乙まで数える

●数える（開始から終了まで）
　制御とは整数
　制御で開始から終了まで繰り返す
　　制御を言う
```

4.3.3　助詞を使わない命令の定義

なでしこでは、助詞を使わずに引数を渡すことのできる命令を定義することも可能です。

そのような命令を定義したいときは、

仮引数名、仮引数名、・・・

という形の仮引数宣言を書きます。つまり、仮引数名を読点（、）で区切って並べるわけです。

このように定義された命令を実行したいときは、

命令名（文、文、・・・）

という形の命令文を書きます。そうすると、命令名の右側の丸括弧の中に書かれたそれぞれの文の値が、引数として命令に渡されます。引数と仮引数とは、命令文の中に並んでいる文の順番と、命令定義の中に並んでいる仮引数名の順番とが一致するように対応づけられます。

プログラムの例　joshinashi.nako

```
甲とは数値　＝　「初項を入力してください。」と尋ねる
乙とは数値　＝　「公差を入力してください。」と尋ねる
丙とは整数　＝　「項数を整数で入力してください。」と尋ねる
等差数列（甲、乙、丙）

●等差数列（初項、公差、項数）
　項とは数値　＝　初項
　項数回
　　項を言う
　　項は項に公差を足す
```

4.4　戻り値を返す命令の定義

4.4.1　戻る文

戻り値を返す命令を定義したいときは、「戻る文」と呼ばれる文を命令定義の中に書きます。

戻る文は、

文で戻る

と書きます。戻る文が実行されると、その中の文が実行されて、その値が戻り値になります。そしてさらに、戻る文は、自分を含んでいる命令の実行を終了させます。

次のプログラムの中で定義されている「三倍」という命令は、1 個の数値を引数として受け取って、それに 3 を掛け算した結果を戻り値として返します。

プログラムの例　sanbai.nako

```
甲とは数値　＝　「数値を入力してください。」と尋ねる
「{甲} の 3 倍は {甲の三倍} です。」と表示
```

```
●三倍（甲の）
  甲に３を掛けるで戻る
```

4.4.2 述語

戻り値として真偽値を返す命令は、「述語」と呼ばれます。

それでは、引数が完全数かどうかを判断する述語を定義してみましょう。

n が自然数だとするとき、n 自身を除いた n のすべての約数の和が n と等しいならば、n は、「完全数」と呼ばれます。たとえば、6 は、$1+2+3=6$ ですから、完全数です。そのほかの完全数としては、28 や 496 や 8128 などがあります。

次のプログラムの中で定義されている完全数という命令は、1 個の自然数を引数として受け取って、それが完全数ならば真、そうでなければ偽を戻り値として返します。

プログラムの例 kanzensuu.nako

```
甲とは整数 ＝ 「自然数を入力してください。」と尋ねる
「{甲} は完全数」と継続表示
もし甲が完全数ならば
  「です。」と表示
違えば
  「ではありません。」と表示

●完全数（甲が）
  和とは整数 ＝ ０
  乙とは整数
  乙で１から（甲から１を引く）まで繰り返す
    もし甲を乙で割った余りが０と等しいならば
      和は和に乙を足す
  甲が和と等しいで戻る
```

4.5 再帰

4.5.1 再帰とは何か

この節では、「再帰」と呼ばれるものについて説明したいと思います。

再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」と形容されます。

ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。

また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

4.5.2 基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。

再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」と呼ばれます。

先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

4.5.3 命令の再帰的な定義

命令は、再帰的に定義することが可能です。命令を再帰的に定義するというのは、定義される当の命令を使って命令を定義するということです。再帰的な構造を持っている概念を取り扱う命令は、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。

命令を再帰的に定義する場合は、それが循環に陥ることを防ぐために、基底について記述した選択肢を作っておくことが必要になります。

4.5.4　再帰による階乗の計算

第 3.4.3 項で説明したように、n の階乗は、

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

という計算をすることによって求めることができるわけですが、これは、

- n が 0 ならば、n の階乗は 1 である。
- n が 0 でないならば、n と、$n-1$ の階乗とを掛け算した結果が n の階乗である。

という再帰的な計算だと考えることができます。

次のプログラムは、階乗を求める「階乗」という命令を再帰的に定義しています。

プログラムの例　kaijou2.nako

```
甲とは整数　＝　「整数を入力してください。」と尋ねる
「{甲} の階乗は {甲の階乗} です。」と表示

●階乗（甲の）
　もし甲が 1 以上ならば
　　甲に甲から 1 を引くの階乗を掛けるで戻る
　違えば
　　1 で戻る
```

4.5.5　再帰による最大公約数の計算

第 3.5.2 項で説明したように、二つの整数の最大公約数は、「ユークリッドの互除法」と呼ばれる方法を使うことによって、簡単に求めることができます。ユークリッドの互助法は、繰り返しを使って説明することもできますが、次のように、再帰的に説明することも可能です。

- m が 0 ならば、n と m の最大公約数は n である。
- m が 0 でないならば、n を m で割った余りを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

次のプログラムは、二つの整数の最大公約数を求める「最大公約数」という命令を再帰的に定義しています。

プログラムの例　gcm2.nako

```
甲とは整数　＝　「1 個目の整数を入力してください。」と尋ねる
乙とは整数　＝　「2 個目の整数を入力してください。」と尋ねる
「{甲} と {乙} の最大公約数は」と継続表示
「{甲と乙の最大公約数} です。」と表示

●最大公約数（甲と乙の）
　もし乙が 0 と等しいならば
　　甲で戻る
　違えば
　　乙と甲を乙で割った余りの最大公約数で戻る
```

第 5 章　文字列

5.1　文字列の基礎

5.1.1　文字列の文字数

この章では、文字列を処理するためのさまざまな命令を紹介したいと思います。

すでに第 1.3.6 項で紹介したように、「文字数」という命令は、引数として文字列を受け取って（助詞は「の」）、その文字列の文字数を戻り値として返します。

プログラムの例　mojisuu2.nako

```
甲とは文字列　＝　「文字列を入力してください。」と尋ねる
```

```
「{甲} の文字数は {甲の文字数} です。」と表示
```

5.1.2 文字列の連結

二つの文字列を左右に並べることによってひとつの文字列を作ることを、文字列を「連結する」と言います。

「&」という命令は、引数として二つの文字列を受け取って、それらの文字列を連結した結果を戻り値として返します。「&」を実行する文は、

$\boxed{文_1}$ & $\boxed{文_2}$

と書きます。そうすると、$文_1$ の値と $文_2$ の値が、引数として「&」に渡されます。たとえば、

「カレー」&「うどん」

という文を実行すると、「カレーうどん」という値が得られます。

プログラムの例 renketsu.nako

```
甲とは文字列 ＝ 「1 個目の文字列を入力してください。」と尋ねる
乙とは文字列 ＝ 「2 個目の文字列を入力してください。」と尋ねる
「{甲} と {乙} とを連結した結果は {甲&乙} です。」と表示
```

5.2 部分文字列

5.2.1 部分文字列の抜き出し

文字列の一部分になっている文字列は、「部分文字列」と呼ばれます。

「文字抜き出す」という命令は、文字列から部分文字列を抜き出して、抜き出した部分文字列を戻り値として返します。「文字抜き出す」を実行する文は、

$\boxed{文_1}$ の $\boxed{文_2}$ から $\boxed{文_3}$ 文字抜き出す

と書きます。$文_1$ のところには部分文字列をそこから抜き出す文字列を求める文、$文_2$ のところには抜き出す部分文字列の位置（先頭から数えた番号）を求める文、$文_3$ のところには抜き出す部分文字列の文字数を求める文を書きます。たとえば、

「いろはにほへとちりぬるを」の 4 から 5 文字抜き出す

という文を実行すると、値として「にほへとち」が得られます。

次のプログラムは、入力された文字列を逆順に並べ替えた文字列を作ります。

プログラムの例 gyakujun.nako

```
甲とは文字列 ＝ 「文字列を入力してください。」と尋ねる
「{甲} を逆順にした結果は {甲の逆順} です。」と表示

●逆順（甲の）
  乙とは文字列 ＝ 「」
  丙とは整数
  丙で甲の文字数から 1 まで繰り返す
    乙は乙&（甲の丙から 1 文字抜き出す）
  乙で戻る
```

5.2.2 左端または右端にある部分文字列の抜き出し

文字列の左端または右端にある部分文字列を抜き出したいときは、「文字抜き出す」を使うことも可能ですが、そのための専用の命令を使うこともできます。

「文字左部分」という命令は、文字列の左端から部分文字列を抜き出して、抜き出した部分文字列を戻り値として返します。「文字左部分」を実行する文は、

$\boxed{文_1}$ の $\boxed{文_2}$ 文字左部分

と書きます。$文_1$ のところには部分文字列をそこから抜き出す文字列を求める文、$文_2$ のところには抜き出す部分文字列の文字数を求める文を書きます。たとえば、

「いろはにほへとちりぬるを」の 5 文字左部分

という文を実行すると、値として「いろはにほ」が得られます。

「文字右部分」という命令は、文字列の右端から部分文字列を抜き出して、抜き出した部分文字列を戻り値として返します。たとえば、

「いろはにほへとちりぬるを」の 5 文字右部分

という文を実行すると、値として「ちりぬるを」が得られます。

次のプログラムは、入力された文字列を中央で分割します（文字数が奇数の場合は、中央の文字の左側で分割します）。

プログラムの例　hanbun.nako

```
甲とは文字列　＝　「文字列を入力してください。」と尋ねる
「{甲} の左半分は {甲の左半分} で、」と表示
「{甲} の右半分は {甲の右半分} です。」と表示

●左半分（甲の）
  半数とは整数　＝　甲の文字数を 2 で割るを切り下げ
  甲の（半数）文字左部分で戻る

●右半分（甲の）
  全数とは整数　＝　甲の文字数
  半数とは整数　＝　全数を 2 で割るを切り下げ
  もし全数を 2 で割った余りが 0 でないならば
    半数は半数に 1 を足す
  甲の（半数）文字右部分で戻る
```

5.3　文字列の探索

5.3.1　文字列の探索の基礎

A と B が文字列だとするとき、A の中で、部分文字列として含まれている B を探し出すことを、A の中で B を「探索する」または「検索する」と言います。

「何文字目」という命令は、文字列の中で左端から右へ向かって文字列を探索して、最初に見つかった部分文字列の位置を戻り値として返します。部分文字列が見つからなかった場合は、0 を返します。

「何文字目」を実行する文は、

$\boxed{文_1}$ で $\boxed{文_2}$ が何文字目

と書きます。文 $_1$ のところには探索する場所となる文字列を求める文、文 $_2$ のところには探索する文字列を求める文を書きます。たとえば、

「いろはにほへとちりぬるを」で「へとちり」が何文字目

という文を実行すると、値として 6 が得られます。

次のプログラムは、漢数字の列を整数に変換するという処理の中で「何文字目」を使っています。

プログラムの例　kansuuji.nako

```
甲とは文字列　＝　「漢数字の列を入力してください。」と尋ねる
「{甲} を整数に変換した結果は {甲の漢数字解析} です。」と表示

●漢数字解析（甲の）
  漢数字とは文字列　＝　「〇一二三四五六七八九」
  乙とは整数　＝　0
  桁とは整数
  丙とは整数
  丙で 1 から甲の文字数まで繰り返す
    桁は漢数字で（甲の丙から 1 文字抜き出す）が何文字目から 1 を引く
    もし桁が-1 でないならば
      乙は乙に 10 を掛けるに桁を足す
  乙で戻る
```

5.3.2 探索を開始する位置の指定

「何文字目」は、左端から探索を開始するわけですが、探索を開始する位置を指定したい場合もあります。そのような場合は、「文字検索」という命令を使います。

「文字検索」は、文字列の中の指定された位置から右へ向かって文字列を探索して、最初に見つかった部分文字列の位置を戻り値として返します。部分文字列が見つからなかった場合は、0を返します。

「文字検索」を実行する文は、

$\boxed{文_1}$で$\boxed{文_2}$から$\boxed{文_3}$を文字検索

と書きます。文$_1$のところには探索する場所となる文字列を求める文、文$_2$のところには探索を開始する位置を求める文、文$_3$のところには探索する文字列を求める文を書きます。たとえば、

「いろはにほへといろはにほへと」で 4 から「はにほ」を文字検索

という文を実行すると、値として 10 が得られます。

次のプログラムは、二つの文字列を読み込んで、1 個目の文字列の中に部分文字列として含まれている 2 個目の文字列の個数を表示します。

プログラムの例 bubunmojiretsusuu.nako

```
甲とは文字列　＝　「文字列を入力してください。」と尋ねる
乙とは文字列　＝　「探索する文字列を入力してください。」と尋ねる
「{甲} の中にある {乙} の個数は」と継続表示
「{甲で乙の部分文字列数} です。」と表示

●部分文字列数（甲で乙の）
  開始位置とは整数　＝　1
  位置とは整数　＝　1
  個数とは整数　＝　0
  位置が 0 でないの間
    位置は甲で開始位置から乙を文字検索
    もし位置が 0 でないならば
      個数は個数に 1 を足す
      開始位置は位置に 1 を足す
  個数で戻る
```

5.4 文字列の置換

5.4.1 文字列の置換の基礎

A、B、C が文字列だとするとき、A の中に部分文字列として含まれている B を C に変更することを、A の中の B を C に「置換する」と言います。

「置換」という命令は、文字列の中で左端から右へ向かって文字列を探索して、部分文字列が見つかるたびにそれを別の文字列に置換する、ということを右端まで繰り返すことによってできる文字列を戻り値として返します。

「置換」を実行する文は、

$\boxed{文_1}$の$\boxed{文_2}$を$\boxed{文_3}$に置換

と書きます。文$_1$のところには探索する場所となる文字列を求める文、文$_2$のところには探索する文字列を求める文、文$_3$のところには見つかった部分文字列をそれに置換する文字列を求める文を書きます。たとえば、

「つき見るつきはこのつきのつき」の「つき」を「月」に置換

という文を実行すると、値として「月見る月はこの月の月」という文字列が得られます。

次のプログラムは、三つの文字列を読み込んで、1 個目の文字列の中に部分文字列として含まれている 2 個目の文字列を 3 個目の文字列に置換した結果を表示します。

プログラムの例 chikan.nako

```
甲とは文字列　＝　「文字列を入力してください。」と尋ねる
乙とは文字列　＝　「探索する文字列を入力してください。」と尋ねる
丙とは文字列　＝　「変更する文字列を入力してください。」と尋ねる
```

```
「{甲} の {乙} を {丙} に置換した結果は」と継続表示
「{甲の乙を丙に置換} です。」と表示
```

5.4.2　最初に見つかった部分文字列だけの置換

「置換」という命令は、見つかった部分文字列をすべて置換することによってできる文字列を戻り値として返すわけですが、最初に見つかった部分文字列だけを置換することによってできる文字列を返す、「単置換」という命令もあります。たとえば、

「つき見るつきはこのつきのつき」の「つき」を「月」に単置換

という文を実行すると、値として「月見るつきはこのつきのつき」という文字列が得られます。

次のプログラムは、先ほどのプログラムの中の「置換」を「単置換」に置換したものです。

プログラムの例　tanchikan.nako

```
甲とは文字列　＝　「文字列を入力してください。」と尋ねる
乙とは文字列　＝　「探索する文字列を入力してください。」と尋ねる
丙とは文字列　＝　「変更する文字列を入力してください。」と尋ねる
「{甲} の {乙} を {丙} に単置換した結果は」と継続表示
「{甲の乙を丙に単置換} です。」と表示
```

第 6 章　配列

6.1　配列の基礎

6.1.1　配列とは何か

なでしこでは、「配列」と呼ばれるデータを扱うことができます。

配列は、データの集合です。配列を構成しているそれぞれのデータは、その配列の「要素」と呼ばれます。

配列を構成している要素は 1 列に並んでいて、それぞれの要素には、それが並んでいる順番のとおりに、「添字」と呼ばれる、0 から始まる番号が与えられています。ですから、個々の要素は添字によって識別することができます。

6.1.2　文字列から配列への変換

配列は、文字列をいくつかの部分に区切って、それらを要素にすることによって作ることができます。

文字列から配列を作りたいときは、「区切る」という命令を使います。「区切る」を実行する文は、

[文 $_1$] を [文 $_2$] で区切る

と書きます。文 $_1$ のところには、要素にしたいデータ（文字列の場合はその文字列、数値の場合はそれをあらわすリテラル）を何らかの文字列で区切って並べた文字列を求める文を書きます。そして、文 $_2$ のところには、それらの文字列またはリテラルを区切っている文字列を求める文を書きます。「区切る」は、文 $_1$ の値を文 $_2$ の値で区切って、それぞれの部分を要素とする配列を戻り値として返します。たとえば、

「53、27、66」を「、」で区切る

という文を実行すると、その値として、53 と 27 と 66 を要素とする配列が得られます。

6.1.3　配列を入れる変数

配列を入れる変数を宣言したいときは、

[変数名] とは配列

という変数宣言を書きます。また、

[変数名] とは配列　＝　[文]

という変数宣言を書くことによって、配列を入れる変数を宣言して、それを配列で初期化する、ということもできます。たとえば、

甲とは配列　＝　「53、27、66」を「、」で区切る

という変数宣言を書くことによって、「甲」という名前の変数を宣言して、53 と 27 と 66 を要素とする配列でそれを初期化することができます。

6.1.4 配列の要素を指定する記述

配列の特定の要素を求めたいときや、特定の要素を変更したいときは、その要素を指定する記述を書きます。

配列の要素を指定する記述は、

[変数名]￥[文]

と書きます。「変数名」のところには、配列が代入されている変数の名前を書きます。そして、「文」のところには、指定したい要素の添字を求める文を書きます。たとえば、

甲￥0

という記述は、「甲」という名前の変数に代入されている配列の中にある先頭の要素を指定します（添字は 0 から始まりますので、先頭の要素の添字は 0 です）。

配列の要素を指定する記述は、文として実行することができます。実行すると、値として、指定された要素が得られます。

配列の特定の要素を変更したいときは、

[配列の要素を指定する記述]は[文]

という代入文を書きます。そうすると、指定された要素が文の値に変更されます。たとえば、

甲￥0 は 80

という代入文は、甲という変数に代入されている配列の先頭の要素を 80 に変更します。

プログラムの例　hairetsu.nako

```
甲とは配列　＝　「53、27、66」を「、」で区切る
甲￥0 は 80
甲￥0 を表示
甲￥1 を表示
甲￥2 を表示
```

6.1.5 配列から文字列への変換

配列は、必要に応じて、暗黙のうちに文字列に変換されます。たとえば、引数として文字列を受け取る命令に配列を渡すと、その配列は、暗黙のうちに文字列に変換されます。

プログラムの例　anmokunohenkan.nako

```
「38、12、54、66、97」を「、」で区切るを表示
```

このプログラムの実行結果から分かるとおり、配列から文字列への暗黙の変換では、配列を構成しているそれぞれの要素は、改行で区切られることになります。改行以外の文字で要素を区切ることによって配列を文字列に変換したいという場合は、暗黙の変換が実行される前に、変換の命令を明示的に実行する必要があります。

区切り文字を指定して配列を文字列に変換したいときは、「配列結合」という命令を使います。

「配列結合」を実行する文は、

[文 $_1$]を[文 $_2$]で配列結合

と書きます。文 $_1$ のところには、配列を求める文を書きます。そして、文 $_2$ のところには、文字列を求める文を書きます。「配列結合」は、文 $_1$ の値を構成しているそれぞれの要素を文 $_2$ の値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。

プログラムの例　hairetsuketsugou.nako

```
「53、72、88、64」を「、」で区切るを「と」で配列結合を表示
```

6.1.6　配列の要素数

配列を構成している要素の個数は、その配列の「要素数」と呼ばれます[1]。

「配列要素数」という命令は、引数として配列を受け取って（助詞は「の」）、その配列の要素数を返します。

プログラムの例　yousosuu.nako

```
「5、3、0、2、6、8、1」を「、」で区切るの配列要素数を表示
```

6.2　配列と命令

6.2.1　引数として配列を受け取る命令

命令は、引数として配列を受け取るように定義することも可能です。

次のプログラムの中で定義されている「配列表示」という命令は、引数として配列を受け取って、その配列を母艦に表示します。

プログラムの例　hairetsuhyouji.nako

```
「27、61、38、44、72」を「、」で区切るを配列表示

●配列表示（甲を）
　甲を「、」で配列結合を表示
```

6.2.2　値呼び出しと参照呼出し

数値や文字列を引数として命令に渡した場合、命令が受け取るのは、その数値や文字列のコピーです。したがって、命令の中で、仮引数に別のデータを代入したとしても、その命令に引数を渡した側にあるデータは変化しません。このような、データのコピーを渡すという引数の渡し方は、「値呼び出し」または「値渡し」と呼ばれます。

それに対して、配列を引数として命令に渡した場合、命令が受け取るのは、その配列のコピーではなくて、その配列の場所をあらわすデータです。したがって、命令の中で、引数として受け取った配列の要素を変更すると、その命令に引数を渡した側にある配列の要素が変化することになります。データの場所をあらわすデータは、「参照」と呼ばれます。このような、データそのものではなくて、データの参照を渡すという引数の渡し方は、「参照呼び出し」または「参照渡し」と呼ばれます。

次のプログラムの中で定義されている「配列変更」という命令は、引数として受け取った配列の要素を変更します。ですから、それを実行すると、引数を渡した側にある配列の要素が変更されます。

プログラムの例　sanshouyobidashi.nako

```
甲とは配列　＝　「89、20、31」を「、」で区切る
甲を配列表示
甲の 1 を 777 に配列変更
甲を配列表示

●配列変更（甲の乙を丙に）
　甲￥乙は丙

●配列表示（甲を）
　甲を「、」で配列結合を表示
```

6.2.3　戻り値として配列を返す命令

命令は、戻り値として配列を返すように定義することも可能です。

次のプログラムの中で定義されている「配列作成」という命令は、配列を作成して、その配列を戻り値として返します。

プログラムの例　hairetsusakusei.nako

[1] 配列の要素数は、「長さ」と呼ばれたり「大きさ」と呼ばれたりすることもあります。

配列作成（10、500）を配列表示

●配列作成（要素数、要素）
甲とは配列
もし要素数が 1 以上ならば
乙とは整数
乙で 0 から（要素数から 1 を引く）まで繰り返す
甲￥乙は要素
甲で戻る

●配列表示（甲を）
甲を「、」で配列結合を表示

6.3 二次元配列

6.3.1 二次元配列の基礎

配列ではないデータ（数値や文字列など）を要素とする配列は、「一次元配列」と呼ばれます。それに対して、一次元配列を要素とする配列は、「二次元配列」または「表」と呼ばれます。

二次元配列の要素の要素になっているそれぞれのデータは、「セル」と呼ばれます。

二次元配列を構成しているそれぞれのセルは、横方向と縦方向に二次元的に並んでいると考えることができます。横方向に並んでいる一列のデータは、「行」と呼ばれます。それに対して、縦方向に並んでいる一列のデータは、「列」と呼ばれます。それぞれの行は、二次元配列の要素になっている一次元配列です。

6.3.2 CSV から二次元配列への変換

コンマと改行を区切り文字列として使って二次元的にデータを並べたものを表現している文字列は、CSV (comma-separated values) と呼ばれます。たとえば、

```
27,54,38
61,30,17
44,92,66
```

というのは CSV の一例です。

「CSV 取得」という命令は、引数として CSV を受け取って（助詞は「を」）、改行を要素の区切り文字列、コンマを要素の要素の区切り文字列とみなして、それを二次元配列に変換します。そして、得られた二次元配列を戻り値として返します。たとえば、

```
「0,0,0,0
0,0,0,0
0,0,0,0」を CSV 取得
```

という文を実行すると、その値として、3 行 4 列で、すべてのセルが 0 の二次元配列が得られます。

6.3.3 二次元配列のセルを指定する記述

二次元配列の特定のセルを求めたいときや、特定のセルを変更したいときは、そのセルを指定する記述を書きます。

二次元配列のセルを指定する記述は、

変数名 ￥ 文 $_1$ ￥ 文 $_2$

と書きます。「変数名」のところには、二次元配列が代入されている変数の名前を書きます。そして、「文 $_1$」のところには指定したいセルを含む行の添字を求める文、「文 $_2$」のところには指定したいセルを含む列の添字を求める文を書きます。たとえば、

```
甲￥2￥3
```

という記述は、「甲」という名前の変数に代入されている二次元配列の中にある 2 行目の 3 列目のセルを指定します（添字は 0 から始まりますので、先頭を 1 と数えると 3 行目の 4 列目）。

二次元配列のセルを指定する記述は、文として実行することができます。実行すると、その値として、指定されたセルが得られます。

二次元配列の特定のセルを変更したいときは、

二次元配列のセルを指定する記述 は 文

という代入文を書きます。そうすると、指定されたセルが文の値に変更されます。たとえば、

甲￥2￥3 は 80

という代入文は、甲という変数に代入されている二次元配列の 2 行目の 3 列目のセルを 80 に変更します。

プログラムの例　nijigenhairetsu.nako

```
甲とは配列　＝　「0,0,0,0
0,0,0,0
0,0,0,0」を CSV 取得
甲￥0￥3 は 678
甲￥2￥1 は 654
甲￥0￥3 を表示
甲￥2￥1 を表示
```

6.3.4　二次元配列から CSV への変換

「表 CSV 変換」という命令は、引数として二次元配列を受け取って（助詞は「を」）、それを CSV に変換した結果を戻り値として返します。

プログラムの例　hyoucsvhenkan.nako

```
甲とは配列　＝　「0,0,0,0
0,0,0,0
0,0,0,0」を CSV 取得
甲￥0￥3 は 678
甲￥2￥1 は 654
甲を表 CSV 変換を表示
```

第 7 章　ハッシュ

7.1　ハッシュの基礎

7.1.1　ハッシュとは何か

なでしこでは、「ハッシュ」と呼ばれるデータを扱うことができます。

配列と同じように、ハッシュも、データの集合です。ハッシュを構成しているそれぞれのデータは、そのハッシュの「要素」と呼ばれます。

ハッシュの 1 個の要素は、「キー」と呼ばれる 1 個の文字列と、「値」と呼ばれる 1 個のデータとを組み合わせたものです。

配列とは違って、ハッシュを構成しているそれぞれの要素は、1 列に並んでいるわけではありません。ですから、添字によって個々の要素を識別するということはできません。

ハッシュを構成しているそれぞれの要素は、その要素のキーによって識別されます。ですから、1 個のハッシュに含まれているそれぞれの要素は、他の要素とは異なるキーを持っている必要があります。

7.1.2　文字列からハッシュへの変換

ハッシュは、文字列をいくつかの部分に区切って、それらを要素にすることによって作ることができます。

文字列からハッシュを作りたいときは、「ハッシュ変換」という命令を使います。「ハッシュ変換」は、ハッシュを記述した文字列を引数として受け取って（助詞は「を」）、それをハッシュに変換した結果を戻り値として返します。

ハッシュを記述した文字列は、ハッシュの要素を記述した、

キー = 値

という形の文字列（イコールは半角）を、改行で区切って並べることによって作ります。たとえば、

```
「国語=73
理科=64
社会=59」をハッシュ変換
```

という文を実行すると、キーが「国語」で値が 73、キーが「理科」で値が 64、キーが「社会」で値が 81 という 3 個の要素から構成されるハッシュが、その値として得られます。

7.1.3 ハッシュを入れる変数

ハッシュを入れる変数を宣言したいときは、

```
変数名 とはハッシュ
```

という変数宣言を書きます。また、

```
変数名 とはハッシュ　＝　文
```

という変数宣言を書くことによって、ハッシュを入れる変数を宣言して、それを初期化することもできます。たとえば、

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
```

という変数宣言を書くことによって、「甲」という名前の変数を宣言して、それをハッシュで初期化することができます。

7.1.4 ハッシュの要素を指定する記述

ハッシュの特定の要素の値を求めたいときや、特定の要素の値を変更したいときは、その要素を指定する記述を書きます。

ハッシュの要素を指定する記述は、

```
変数名 @ 文
```

と書きます。「変数名」のところには、ハッシュが代入されている変数の名前を書きます。そして、「文」のところには、指定したい要素のキーを求める文を書きます。たとえば、

```
甲@「国語」
```

という記述は、「甲」という名前の変数に代入されているハッシュの中にある、「国語」をキーとする要素を指定します。

ハッシュの要素を指定する記述は、文として実行することができます。実行すると、指定された要素の値が、その値として得られます。

ハッシュの特定の要素について、その値を変更したいときは、

```
ハッシュの要素を指定する記述 は 文
```

という代入文を書きます。そうすると、指定された要素の値が文の値に変更されます。たとえば、

```
甲@「国語」は 80
```

という代入文は、甲という変数に代入されているハッシュの中にある、「国語」をキーとする要素の値を 80 に変更します。

プログラムの例　hash.nako

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
甲@「国語」は 80
甲@「国語」を表示
甲@「理科」を表示
甲@「社会」を表示
```

7.1.5　ハッシュから文字列への変換

引数として文字列を受け取る命令に、引数としてハッシュを渡すと、そのハッシュは暗黙のうちに文字列に変換されます。

たとえば、「表示」という命令に引数としてハッシュを渡すと、そのハッシュは暗黙のうちに文字列に変換されますので、その文字列が母艦の上に表示されることになります。

プログラムの例　hashhyouji.nako

```
「題名=テキサスの五人の仲間
監督=フィルダー・クック
脚本=シドニー・キャロル」をハッシュ変換を表示
```

7.2　要素の追加と削除

7.2.1　ハッシュへの要素の追加

ハッシュに対しては、要素をいくらでも追加することができます。

第 7.1.4 節で説明したように、ハッシュの特定の要素について、その値を変更したいときは、

[ハッシュの要素を指定する記述]は[文]

という代入文を実行すればいいわけですが、この形の代入文は、要素の値を変更したいときだけではなくて、ハッシュに要素を追加したいときにも使うことができます。この形の代入文を実行したときに要素の値が変更されるのは、指定されたキーを持つ要素がすでに存在している場合だけです。指定されたキーを持つ要素が存在していなかった場合は、そのキーを持つ新しい要素が追加されることになります。ですから、ハッシュに要素を追加したいときは、まだ存在していないキーを持つ要素を指定する代入文を実行すればいいわけです。

プログラムの例　yousonotsuika.nako

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
甲@「算数」は 88
甲を表示
```

7.2.2　ハッシュからの要素の削除

ハッシュからは、要素を削除することもできます。

ハッシュから要素を削除したいときは、「ハッシュキー削除」という命令を使います。この命令は、引数としてハッシュ（助詞は「の」）とキー（助詞は「を」）を受け取って、そのハッシュから、キーで指定された要素を削除します。

プログラムの例　yousonosakujo.nako

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
甲の「理科」をハッシュキー削除
甲を表示
```

7.3　反復文

7.3.1　反復文の基礎

ハッシュを構成しているすべての要素に対して何らかの処理を実行したい、というときは、「反復文」という文を使います。

反復文は、

[文]を反復

　[プログラム]

と書きます。この中の「文」のところには、ハッシュを求める文を書きます。反復文の中のプログラムは、インデントが必要です。

反復文は、指定されたハッシュを構成しているすべての要素について、その要素をあらわす、

キー=値

という形の文字列（イコールは半角）を「それ」という変数に代入して、指定されたプログラムを実行する、ということを繰り返す、という動作をあらわしています。

プログラムの例 hanpukubun.nako

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
要素とは配列
甲を反復
  要素はそれを「=」で区切る
  「キーが {要素￥0} で値が {要素￥1} の要素があります。」と表示
```

7.3.2 反復文の中の回数

回文や繰り返す文や間文の場合と同じように、反復文の中でも、「回数」という変数を使うことができます。反復文の場合も、この変数には、繰り返しが何回目なのかということを示す整数が代入されます。

プログラムの例 kaisuu3.nako

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
甲を反復
  「要素は {それ} で、回数は {回数} です。」と表示
```

7.4 ハッシュと命令

7.4.1 引数としてハッシュを受け取る命令

命令は、引数としてハッシュを受け取るように定義することも可能です。

ハッシュを引数として命令に渡すときの渡し方は、配列の場合と同じように、参照呼び出しです。

次のプログラムの中で定義されている「ハッシュ加算」という命令は、引数としてハッシュ（助詞は「に」）と数値（助詞は「を」）を受け取って、そのハッシュが持っているすべての要素の値に対して、その数値を加算します。

プログラムの例 hashkasan.nako

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
甲を表示
甲に 20 をハッシュ加算
甲を表示

●ハッシュ加算（甲に乙を）
  要素とは配列
  甲を反復
    要素はそれを「=」で区切る
    甲@（要素￥0）は要素￥1 に乙を足す
```

7.4.2 戻り値としてハッシュを返す命令

命令は、戻り値としてハッシュを返すように定義することも可能です。

次のプログラムの中で定義されている「ハッシュ合成」という命令は、引数として二つのハッシュを受け取って（助詞は「と」と「の」）、それらのハッシュを合成してできた新しいハッシュを戻り値として返します。

プログラムの例　hashgousei.nako

```
甲とはハッシュ　＝　「国語=73
理科=64
社会=59」をハッシュ変換
乙とはハッシュ　＝　「算数=46
音楽=82
図工=97」をハッシュ変換
甲と乙のハッシュ合成を表示

●ハッシュ合成（甲と乙の）
　丙とはハッシュ　＝　甲
　要素とは配列
　乙を反復
　　要素はそれを「=」で区切る
　　丙@（要素￥0）は要素￥1
　丙で戻る
```

第 8 章　グループ

8.1　グループの基礎

8.1.1　グループとは何か

なでしこでは、「グループ」と呼ばれるものを扱うことができます。

グループというのは、いくつかの変数または命令から構成される特殊な変数だと考えることができます。

グループを構成している変数または命令は「メンバー」と呼ばれます。それぞれのメンバーは、「メンバー名」と呼ばれる識別子によって識別されます。

グループのメンバーになっている変数は「メンバー変数」と呼ばれ、グループのメンバーになっている命令は「メンバー命令」または「メソッド」と呼ばれます。

8.1.2　グループ型

グループは、「グループ型」と呼ばれる型を持ちます。

グループは、どのようなメンバーを持つのかということによって、異なるグループ型を持つことになります。ですから、グループ型というのは、ひとつの型ではなくて、さまざまな型の総称です。

特定のグループ型は、識別子によって識別されます。グループ型に与えられた識別子は、「グループ型名」と呼ばれます。

8.1.3　グループ型定義

なでしこはさまざまなグループ型を持っていますが、プログラムの中でグループ型を定義するということも可能です。グループ型を定義したいときは、「グループ型定義」と呼ばれる記述をプログラムの中に書きます。

グループ型定義は、

■ グループ型名
　メンバー定義
　　⋮

と書きます。「グループ型名」のところには、定義されるグループ型に名前として与えたい識別子を書きます。

グループ型定義の中には、「メンバー定義」と呼ばれるものを何個でも書くことができます。メンバー定義は、インデントが必要です。

メンバー定義は、その名前のとおり、メンバーを定義する記述です。メンバー変数を定義するメンバー定義は「メンバー変数定義」と呼ばれ、メンバー命令を定義するメンバー定義は「メンバー命令定義」と呼ばれます。

メンバー変数定義は、

・ メンバー変数名

と書きます。「メンバー変数名」のところには、定義されるメンバー変数に名前として与えたい識別子を書きます。

たとえば、「名前」と「年齢」という二つのメンバー変数定義から構成される、「人間」というグループ型名を持つグループ型を定義したいときは、

■人間
・名前
・年齢

というグループ型定義を書きます。

メンバー命令定義については、第 8.2 節で説明することにしたいと思います。

8.1.4 グループの宣言

グループを作ることを、グループを「宣言する」と言います。

グループを宣言するという動作は、「グループ宣言」と呼ばれるものによって記述されます。

グループ宣言は、

識別子 とは グループ型名

と書きます。「グループ型名」のところには、宣言したいグループのグループ型名を書きます。

グループ宣言は、グループを宣言して、その名前として識別子を与えます。たとえば、

甲とは人間

というグループ宣言は、「人間」というグループ型のグループを作って、それに対して「甲」という名前を与えます。

グループ型定義と、それによって定義されたグループ型のグループを宣言するグループ宣言との位置関係は、どちらが上でどちらが下でもかまいません。

8.1.5 グループのメンバー変数を指定する記述

グループの特定のメンバー変数に代入されているデータを求めたいときや、特定のメンバー変数にデータを代入したいときは、そのメンバー変数を指定する記述を書きます。

グループのメンバー変数を指定する記述は、

グループ名 の メンバー変数名

と書きます。「グループ名」のところにはグループの名前を書いて、「メンバー変数名」のところには、指定したいメンバー変数の名前を書きます。たとえば、

甲の名前

という記述は、「甲」というグループの中にある「名前」というメンバー変数を指定します。

グループのメンバー変数を指定する記述は、文として実行することができます。実行すると、値として、指定されたメンバー変数に代入されているデータが得られます。

グループの特定のメンバー変数にデータを代入したいときは、

グループのメンバー変数を指定する記述 は 文

という代入文を書きます。そうすると、指定されたメンバー変数に文の値が代入されます。たとえば、

甲の名前は「鹿目まどか」

という代入文は、甲というグループの中にある「名前」というメンバー変数に、「鹿目まどか」という文字列を代入します。

プログラムの例 group.nako

```
■人間
・名前
・年齢
```

```
甲とは人間
甲の名前は「鹿目まどか」
甲の年齢は 14
甲の名前を表示
甲の年齢を表示
```

8.1.6　について文

ひとつのグループについて、そのグループが持っているさまざまなメンバーを扱う処理を記述するとき、「同じグループの名前を何回も書かないといけないというのはわずらわしすぎる」と感じることがあります。そんなときは、「について文」という文を使うといいでしょう。この文を使うと、グループ名を省略して、メンバー名だけでメンバーを指定することができます。

について文は、

　グループ名 について

　　プログラム

と書きます。「グループ名」のところには、そのメンバーを扱いたいグループの名前を書きます。プログラムは、インデントが必要です。

について文は、その中のプログラムを実行するという動作を意味しています。そのプログラムの中では、について文の冒頭で指定されたグループに関しては、グループ名を省略して、メンバー名だけでメンバーを指定することができます。

プログラムの例　nitsuite.nako

```
■人間
  ・名前
  ・年齢

甲とは人間
甲について
  名前は「暁美ほむら」
  年齢は 14
  名前を表示
  年齢を表示
```

8.2　メンバー命令

8.2.1　メンバー命令の定義

第 8.1 節で説明したように、グループは、「メンバー」と呼ばれる変数または命令から構成されます。グループのメンバーになっている変数は「メンバー変数」と呼ばれ、グループのメンバーになっている命令は「メンバー命令」と呼ばれます。

メンバー命令を持つメンバーを作るためには、グループ型定義の中に、メンバー命令を定義するメンバー定義を書く必要があります。第 8.1.3 項で説明したように、メンバー命令を定義するメンバー定義は、「メンバー命令定義」と呼ばれます。

引数を受け取らないメンバー命令を定義するメンバー命令定義は、

　・ メンバー命令名 ～

　　プログラム

と書きます。「メンバー命令名」のところには、定義されるメンバー命令に名前として与えたい識別子を書きます。プログラムは、インデントが必要です。プログラムが 1 行だけの場合は、

　・ メンバー命令名 ～ プログラム

というように、1 行にまとめて書くこともできます。

8.2.2　メンバー命令の実行

メンバー命令を実行したいときは、そのメンバー命令を実行する命令文を書きます。

グループのメンバー命令を実行する命令文は、そのメンバー命令が引数を受け取らないならば、

グループ名 の メンバー命令名

と書きます。「グループ名」のところにはグループの名前を書いて、「メンバー命令名」のところには、実行したいメンバー命令の名前を書きます。たとえば、

甲のあいさつ

という記述は、「甲」というグループの中にある「あいさつ」というメンバー命令を実行します。

プログラムの例　membermeirei.nako

```
■グループ甲
・あいさつ〜「こんにちは。」と表示

甲とはグループ甲
甲のあいさつ
```

8.2.3　引数を受け取るメンバー命令の定義と実行

普通の命令と同じように、メンバー命令も引数を受け取ることができます。引数を受け取るメンバー命令を定義したい場合は、普通の命令の場合と同じように、メンバー命令名の右側に丸括弧を書いて、その丸括弧の中に仮引数宣言を書きます。

引数を受け取る普通の命令を実行する場合と同じように、引数を受け取るメンバー命令を実行したい場合も、引数として渡したいデータを求める文を命令文の中に書きます。

プログラムの例　membermeireihikisuu.nako

```
■グループ甲
・あいさつ（相手に）〜「{相手} さん、こんにちは。」と表示

甲とはグループ甲
「友子」に甲のあいさつ
```

8.2.4　戻り値を返すメンバー命令の定義と実行

普通の命令と同じように、メンバー命令も戻り値を返すことができます。戻り値を返すメンバー命令を定義したい場合は、普通の命令の場合と同じように、メンバー命令定義の中に戻る文を書きます。

戻り値を返す普通の命令を実行した場合と同じように、戻り値を返すメンバー命令を実行した場合も、そのメンバー命令が返した戻り値は、そのメンバー命令を実行した命令文の値になります。

プログラムの例　membermeireimodorichi.nako

```
■グループ甲
・三倍（甲の）〜甲に 3 を掛けるで戻る

甲とはグループ甲
7 の甲の三倍を表示
```

8.2.5　メンバー命令定義の中でのメンバーの指定

グループのメンバーを指定するためには、原則的には、メンバー名だけではなくてグループ名も必要です。しかし、メンバー命令定義の中で、自分と同じグループの中にあるメンバーを指定するときは、グループ名を書かないで、メンバー名だけを書きます。

プログラムの例　membermeireimember.nako

```
■人間
・名前
・名前表示〜「私の名前は {名前} です。」と表示

甲とは人間
甲の名前は「早乙女和子」
甲の名前表示
```

8.2.6　コンストラクター

グループを宣言したときに自動的に実行される命令は、「コンストラクター」と呼ばれます。

グループの中に、「作る」という名前のメンバー命令を定義しておくと、それはコンストラクターになります。つまり、グループを宣言したとき、そのグループが「作る」という名前のメンバー命令を持っているならば、それが自動的に実行されます。

プログラムの例　tsukuru.nako

```
■財布
　・金額
　・作る～金額は 1000

甲とは財布
甲の金額を表示
```

8.3　グループ型のミックス

8.3.1　グループ型のミックスの基礎

なでしこには、グループ型を定義するとき、すでに定義されている別のグループ型のメンバー定義をその中へ追加することができる、という機能があります。この機能を使って、グループ型に別のグループ型のメンバー定義を追加することを、グループ型を「ミックスする」と言います。

グループ型をミックスすることによって新しいグループ型を定義するというのは、すでに定義されているグループ型を特殊化した新しいグループ型を定義することだと考えることができます。

何らかの類似点を持っているいくつかのグループ型を定義する必要がある場合は、まず、それらを一般化したグループ型を作って、次に、それをミックスした個々のグループ型を定義する、というようにグループ型を階層化することによって、プログラムをすっきりと記述することができます。

8.3.2　ミックスを伴うグループ型の定義

新しいグループ型を定義するときに、すでに定義されている別のグループ型をそこへミックスしたい場合は、

■ グループ型名 $_1$ ＋ グループ型名 $_2$

　メンバー定義

　　⋮

という形のグループ型定義を書きます。グループ型名 $_1$ のところには、新しく定義されるグループ型に与えたい識別子を書いて、グループ型名 $_2$ のところには、そのグループ型にミックスしたい既存のグループ型の名前を書きます。たとえば、「人間」というグループ型がすでに定義されているとするとき、

```
■力士＋人間
　・四股名
```

というグループ型定義を書くことによって、「人間」をミックスした「力士」というグループ型を定義することができます。グループ型をミックスするというのは、そのグループ型を構成しているすべてのメンバー定義を追加するということですから、「力士」というグループ型は、「人間」が持っているすべてのメンバー定義と、「四股名」というメンバー変数の定義から構成されることになります。

プログラムの例　mix.nako

```
■人間
　・名前

■力士＋人間
　・四股名
```

```
甲とは力士
甲について
  名前は「田中太郎」
  四股名は「生駒山」
  名前を表示
  四股名を表示
```

8.3.3　ミックスによる名前の衝突

同一の名前を持つものが複数個存在していて、その名前ではそれらを識別することができない状態は、名前の「衝突」と呼ばれます。

グループ型のミックスという機能を使う場合も、名前の衝突が起きる可能性があります。定義される新しいグループ型と、それにミックスされる既存のグループ型の両方に、同じ名前を持つメンバーの定義が存在していた場合です。

そのような場合は、いったいどうなるのでしょうか。どうなるかというのは、試してみれば一目瞭然です。次のプログラムを実行してみましょう。

プログラムの例　shoutotsu.nako

```
■グループ甲
  ・同名命令～「私は甲のメンバー命令です。」と表示

■グループ乙＋グループ甲
  ・同名命令～「私は乙のメンバー命令です。」と表示

乙とはグループ乙
乙の同名命令
```

実行すると、「私は乙のメンバー命令です。」と表示されるはずです。つまり、グループ型のミックスによって名前が衝突した場合は、ミックスされる既存のグループ型のメンバー定義が隠されて、新しいグループ型のメンバー定義だけが有効になるということです。

第 9 章　GUI の基礎

9.1　母艦

9.1.1　この章について

コンピュータとそのユーザーとのあいだのインターフェースで、画面の上に表示されたグラフィックスと、マウスなどのポインティングデバイスを使うものは、「GUI」と呼ばれます（GUI は、「グラフィカルユーザーインターフェース」の略語です）。

この章では、なでしこで GUI を扱うための基礎的な知識について説明したいと思います。

9.1.2　GUI 部品

GUI は、画面上に表示されるさまざまな視覚的な部品を組み合わせることによって作られます。そのような視覚的な部品は、「GUI 部品」と呼ばれます。

なでしこのプログラムの中では、個々の GUI 部品は、それに対応するグループとして扱われます。

9.1.3　母艦の基礎

第 1.2.3 項で説明したように、なでしこのプログラムを実行したときに開かれるウィンドウは、「母艦」と呼ばれます。

母艦は、なでしこのプログラムそのものを視覚化したものだと考えることができます。ですから、母艦を閉じるという操作は、なでしこのプログラムを終了させるという意味になります。

母艦というのも GUI 部品のひとつですので、プログラムの中では、それに対応するグループとして扱われることになります。GUI 部品に対応するグループは、原則的にはプログラムの中で宣言することが必要ですが、母艦だけは宣言する必要がありません。母艦に対応するグループは、「母艦」という名前です。

9.1.4　母艦のタイトル

母艦は、「タイトル」というメンバー変数を持っています。このメンバー変数には、母艦のタイトルバーに表示されるタイトルが代入されています。このメンバー変数に文字列を代入することによって、母艦のタイトルを変更することも可能です。タイトルのデフォルトは「なでしこ」です。

次のプログラムは、母艦のタイトルを「私は母艦です。」に変更します。

プログラムの例　bokantitle.nako

```
母艦のタイトルは「私は母艦です。」
```

9.1.5　母艦の大きさ

ウィンドウから外枠やタイトルバーを取り除いた残りの部分は、「クライアント領域」と呼ばれます。

母艦は、「クライアント幅」というメンバー変数と「クライアント高さ」というメンバー変数を持っています。これらのメンバー変数には、母艦のクライアント領域の大きさが代入されています。クライアント幅は横の長さ、クライアント高さは縦の長さで、単位はピクセルです。これらのメンバー変数に整数を代入することによって、母艦のクライアント領域の大きさを変更することも可能です。これらのメンバー変数のデフォルトは、クライアント幅が 640 ピクセルで、クライアント高さが 400 ピクセルです。

次のプログラムは、母艦のクライアント領域の横の長さを 800 ピクセル、縦の長さを 600 ピクセルに変更します。

プログラムの例　bokanookisa.nako

```
母艦のクライアント幅は 800
母艦のクライアント高さは 600
```

9.2　定型的なダイアログボックス

9.2.1　定型的なダイアログボックスの基礎

なでしこでは、命令を実行することによって、定型的なダイアログボックスを表示することができます。

定型的なダイアログボックスを表示する命令としては、すでに第 1.5 節で、「言う」と「尋ねる」という二つのものを紹介しましたが、なでしこには、それら以外にも次のような命令があります。

二択	「はい」または「いいえ」で答えることのできる質問をするダイアログボックスを表示する命令。
三択	「はい」「いいえ」「キャンセル」のいずれかで答えることのできる質問をするダイアログボックスを表示する命令。
リスト選択	任意の個数の選択肢からひとつを選ぶダイアログボックスを表示する命令。
メモ記入	文字列の編集をするダイアログボックスを表示する命令。

この節では、これらの命令について説明したいと思います。

9.2.2　二択

「二択」は、質問となる文字列を引数として受け取って（助詞は「と」）、「はい」と「いいえ」という二つのボタンを持つダイアログボックスを表示します。戻り値は、クリックされたボタンが「はい」ならば 1、「いいえ」ならば 0 です。

プログラムの例　nitaku.nako

```
もし「あなたはプログラミングが好きですか？」と二択ならば
  「それは素晴らしい。」と表示
違えば
  「それは残念ですね。」と表示
```

9.2.3 三択

「三択」は、質問となる文字列を引数として受け取って（助詞は「と」）、「はい」と「いいえ」と「キャンセル」という三つのボタンを持つダイアログボックスを表示します。戻り値は、クリックされたボタンが「はい」ならば 1、「いいえ」ならば 0、「キャンセル」ならば 2 です。

なでしこは、1、0、2 のそれぞれで初期化された「はい」「いいえ」「キャンセル」という変数をあらかじめ定義していますので、「三択」の戻り値で動作を選択する条件分岐文は、これらの変数を使って書くことができます。

プログラムの例　santaku.nako

```
「この内容で送信していいですか？」と三択で条件分岐
  はいならば
    「送信しました。」と表示
  いいえならば
    「内容を変更してください。」と表示
  キャンセルならば
    「送信をキャンセルしました。」と表示
```

9.2.4 リスト選択

「リスト選択」は、選択肢のそれぞれを改行で区切って並べた文字列、または選択肢のそれぞれを要素とする配列を引数として受け取って（助詞は「から」）、それらの選択肢からひとつを選択するか、または選択肢にはない文字列を入力するダイアログボックスを表示します。戻り値は、「OK」というボタンがクリックされた場合は選択された選択肢または入力された文字列で、「キャンセル」というボタンがクリックされた場合は空文字列です。

プログラムの例　listsentaku.nako

```
選択結果とは文字列　＝　「コーヒー
紅茶
ジュース」からリスト選択
もし選択結果が「」ならば
  「選択はキャンセルされました。」と表示
違えば
  「あなたが選んだのは {選択結果} です。」と表示
```

9.2.5 メモ記入

「メモ記入」は、文字列を引数として受け取って（助詞は「を」）、その文字列を編集するダイアログボックスを表示します。戻り値は、「決定」というボタンがクリックされた場合は編集後の文字列で、「取消」というボタンがクリックされた場合は空文字列です。

プログラムの例　memokinyuu.nako

```
編集結果とは文字列　＝　「この文章を
自由に
編集してください。」をメモ記入
もし編集結果が「」ならば
  「編集は取り消されました。」と表示
違えば
  編集結果を表示
```

9.3 イベント

9.3.1 イベントの基礎

ユーザーによる操作などによって発生する出来事は、「イベント」と呼ばれます。

GUI を持つプログラムは、常にイベントの発生を待っていて、イベントが発生すると、それに対応する処理を実行するように作る必要があります。イベントが発生したときにプログラムによって実行される処理は、「イベント処理」と呼ばれます。「この GUI 部品でこのようなイベントが発生したときはこのようなイベント処理を実行する」という動作を記述することを、イベント処理を「定義する」と言います。

9.3.2　イベント定義

なでしこでイベント処理を定義したいときは、「イベント定義」と呼ばれるものを書きます。イベント定義は文の一種で、イベント処理を定義するという動作を意味しています。

イベント定義は、

[グループ名]の[イベント名]は
　[プログラム]

と書きます。イベント定義の中のプログラムは、インデントが必要です。

「グループ名」のところには、GUI 部品のグループ名を書きます。そうすると、その GUI 部品でイベントが発生したときのイベント処理が定義されることになります。たとえば、グループ名として「母艦」と書くことによって、母艦で発生したイベントを処理するイベント処理を定義することができます。

「イベント名」のところには、定義の対象となるイベントの種類をあらわす名前を書きます。たとえば、「クリックした時」というイベント名を書くことによって、マウスの左ボタンでクリックされたときに実行されるイベント処理を定義することができます。

プログラムの例　click.nako

```
母艦のクリックした時は
  「母艦がクリックされました。」と言う。
```

イベント定義は、その中のプログラムが 1 行だけの場合、

[グループ名]の[イベント名]は〜[プログラム]

というように 1 行で書くことも可能です。

9.3.3　マウスによるイベント

マウスが操作されたときに実行されるイベント処理を定義するために使われるイベント名としては、次のようなものがあります。

クリックした時	マウスの左ボタンでクリックした。
ダブルクリックした時	マウスの左ボタンでダブルクリックした。
マウス押した時	マウスのボタンを押した。
マウス離した時	マウスのボタンを離した。
マウス移動した時	マウスが移動した。

プログラムの例　doubleclick.nako

```
母艦のダブルクリックした時は
  「母艦がダブルクリックされました。」と言う。
```

9.3.4　マウスの位置

マウスによるイベントが発生すると、そのときのマウスの位置が、GUI 部品が持っている次の二つのメンバー変数に代入されます。

マウス X　GUI 部品の左端からマウスの位置までの水平方向の距離。単位はピクセル。
マウス Y　GUI 部品の上端からマウスの位置までの垂直方向の距離。単位はピクセル。

ただし、「クリックした時」と「ダブルクリックした時」というイベントでは、これらのメンバー変数にマウスの位置は代入されません。マウスがクリックされたときのマウスの位置を取得したい場合は、「マウス離した時」を使う必要があります。

次のプログラムは、母艦の上でマウスを移動させると、そのときのマウスの位置をタイトルバーに表示します。

プログラムの例　mouseichi.nako

```
母艦のマウス移動した時は
  母艦のタイトルは「( {母艦のマウス X} ,  {母艦のマウス Y} )」
```

9.3.5 マウスのボタンの識別

「マウス押した時」または「マウス離した時」というイベントが発生すると、そのときに操作されたボタンを識別するための文字列が「押されたボタン」というメンバー変数に代入されます。ボタンを識別するための文字列は、左ボタンは「左」、右ボタンは「右」、中央ボタンは「中央」です。

プログラムの例 `mousebutton.nako`

```
母艦のマウス離した時は
  「それは{母艦の押されたボタン}のボタンです。」と言う。
```

9.3.6 キーボードのイベント

イベント定義の中に次のようなイベント名を書くことによって、キーボードのキーに対する操作によって発生したイベントを処理するイベント処理を記述することができます。

キー押した時	キーボードのキーを押した。
キータイピング時	キーボードの文字キーを押した。
キー離した時	キーボードのキーを離した。

プログラムの例 `keyevent.nako`

```
母艦のキー押した時は
  「キーボードのキーが押されました。」と言う。
```

9.3.7 キーボードのキーの識別

「キー押した時」または「キー離した時」というイベントが発生すると、そのときに操作されたキーを識別するための整数が「押された仮想キー」というメンバー変数に代入されます。

プログラムの例 `kasoukey.nako`

```
母艦のキー押した時は
  「それは{母艦の押された仮想キー}のキーです。」と言う。
```

9.3.8 キーボードの文字キーの識別

「キータイピング時」というイベントが発生すると、そのときに操作された文字キーを識別するための文字（文字数が 1 の文字列）が「押されたキー」というメンバー変数に代入されます。

プログラムの例 `mojikey.nako`

```
母艦のキータイピング時は
  「それは{母艦の押されたキー}のキーです。」と言う。
```

9.4 タイマー

9.4.1 タイマーの基礎

GUI を持つプログラムは、基本的には、ユーザーが何も操作していないときは何もしないで待機しているだけですが、場合によっては、ユーザーが何も操作していないときに何らかの動作を実行することが必要になることもあります。

ユーザーによる操作とは無関係に動作を実行するプログラムをなでしこで書きたいときは、「タイマー」というグループ型名の GUI 部品を使います。

タイマーは、「時満ちた時」というイベント名で識別されるイベントを、一定の時間間隔で発生させます。ですから、そのイベントが発生したときに実行されるイベント処理を定義することによって、ユーザーとは無関係に動作を実行するプログラムを書くことができます。

GUI 部品は、何らかのグラフィックスとして画面の上に表示されるのですが、タイマーはその例外で、GUI 部品であるにもかかわらず、画面の上には何も表示されません。

9.4.2 時間間隔の設定

タイマーを使うためには、それを宣言したのち、イベントを発生させる時間間隔を設定する必要があります。時間間隔は、タイマーが持っている「値」または「間隔」というメンバー変数に

数値で設定します。「値」と「間隔」の相違点は時間の単位です。「値」に設定した数値は単位がミリ秒だと解釈されて、「間隔」に設定した数値は単位が秒だと解釈されます。

9.4.3　タイマーの開始と停止

タイマーは、それが持っているメンバー命令を実行することによって、開始させたり停止させたりすることができます。タイマーを開始させるメンバー命令は「開始」で、停止させるメンバー命令は「停止」です。

タイマーは、宣言した直後は停止しています。ですから、それを使うときには、最初に「開始」を実行する必要があります。

次のプログラムは、母艦のタイトルバーに表示された整数を 100 ミリ秒ごとに 1 だけ増やします。

プログラムの例　timer.nako

```
甲とは整数　＝　0
乙とはタイマー
乙について
  値は 100
  時満ちた時は
    甲は甲に 1 を足す
    母艦のタイトルは甲
  開始
```

第 10 章　GUI 部品

10.1　GUI 部品の基礎

10.1.1　GUI 部品の座標系

GUI 部品の大多数は、別の GUI 部品を土台として、その上に配置されます。土台となっている GUI 部品は、その上に配置される GUI 部品の「親」と呼ばれます。

GUI 部品を配置する位置は、その親の座標系を使って指定する必要があります。

GUI 部品の座標系は、その左上の隅が原点になっています（ただし、ウィンドウの座標系の原点は、ウィンドウ全体の左上の隅ではなくて、クライアント領域の左上の隅です）。そして、x 軸は右向きで、y 軸は下向きです。

GUI 部品の座標系では、距離の単位としてピクセルを使います。

10.1.2　GUI 部品の位置と大きさ

GUI 部品の位置と大きさは、次のメンバー変数にデータを代入することによって設定することができます。

位置　GUI 部品の左上の隅の位置。x 座標と y 座標をコンマで区切って並べた文字列で指定する。たとえば、「300,200」は、x 座標が 300 で y 座標が 200 という意味。

幅　GUI 部品の横の長さ。

高さ　GUI 部品の縦の長さ。

それでは、「プッシュボタン」と呼ばれる GUI 部品を使って、GUI 部品の位置と大きさを設定するプログラムを書いてみましょう（プッシュボタンという GUI 部品の本来の使い方については、第 10.2.2 項で説明します）。

プッシュボタンは、「ボタン」というグループ型から作ることができます。

プッシュボタンが持っている「位置」、「幅」、「高さ」というメンバー変数には、あらかじめ初期値が代入されていますが、その内容は、プログラムの中でそれらにデータを代入することによって自由に変更することができます。

プログラムの例　ichitoookisa.nako

```
甲とはボタン
甲について
  位置は「40,20」
  幅は 80
  高さは 300
```

```
乙とはボタン
乙について
  位置は「160,200」
  幅は 300
  高さは 80
```

10.1.3 GUI 部品の配置

GUI 部品は、デフォルトでは、直前に作られた GUI 部品の下の位置に配置されます。ですから、いくつかの GUI 部品を縦に並べる場合は、それらを上から順番に作っていけば、その位置を明示的に指定する必要はありません。

GUI 部品を左右に並べたいときは、明示的に座標を指定するか、「右側」というメンバー命令を使います。

GUI 部品の多くは、「右側」というメンバー命令を持っています。このメンバー命令は、自分の右側に GUI 部品を配置するための適切な位置をあらわす文字列を戻り値として返します。ですから、既存の GUI 部品の「右側」が返した戻り値を新しい GUI 部品の「位置」に代入することによって、既存の GUI 部品の右側に新しい GUI 部品を配置することができます。

プログラムの例 migigawa.nako

```
甲とはボタン
乙とはボタン
乙の位置は甲の右側
```

10.1.4 GUI 部品のテキスト

GUI 部品の多くは、「テキスト」というメンバー変数を持っています。そして、多くの GUI 部品は、自分が持っているテキストを自分の上に表示します。

プッシュボタンが持っている「幅」というメンバー変数には、テキストを表示するのに十分な長さを代入する必要があります。

プログラムの例 text.nako

```
甲とはボタン
甲について
  幅は 360
  テキストは「このテキストはボタンの上に表示されます。」
```

10.2 ボタン

10.2.1 ボタンの基礎

この節では、「ボタン」と呼ばれる GUI 部品について説明したいと思います。

「ボタン」という言葉には、広い意味と狭い意味があります。広い意味の「ボタン」は、次の 3 種類の GUI 部品の総称です。

- プッシュボタン
- チェックボックス
- ラジオボタン

プッシュボタンは、単に「ボタン」と呼ばれることもあります。つまり、プッシュボタンというのが、「ボタン」という言葉の狭い意味です。

10.2.2 プッシュボタン

クリックされたときに何らかの動作をする、という目的で使われる GUI 部品は、「プッシュボタン」と呼ばれます。プッシュボタンは、「ボタン」というグループ型から作ることができます。

ボタンがクリックされたときに実行されるイベント処理は、「クリックした時」というイベントを使って定義します。

プログラムの例 button.nako

```
甲とはボタン
甲について
  幅は 220
```

```
  テキストは「私をクリックしてください。」
  クリックした時は
    「ボタンがクリックされました。」と言う
```

10.2.3　チェックボックス

クリックによってチェックを入れたりはずしたりすることのできる GUI 部品は、「チェックボックス」と呼ばれます。チェックボックスは、「チェック」というグループ型から作ることができます。

チェックボックスにチェックが入っているか入っていないかという状態は、「値」というメンバー変数の内容を調べることによって判定することができます。「値」の内容は、チェックが入っている場合は 1 で、入っていない場合は 0 です。

チェックボックスが持っている「高さ」というメンバー変数には、適切な縦の長さが自動的に代入されます。しかし、「幅」というメンバー変数には、テキストを表示するのに十分な長さを代入する必要があります。

プログラムの例　check.nako

```
甲とはチェック
甲について
  幅は 200
  テキストは「私はチェックボックスです。」
乙とはボタン
乙について
  幅は 200
  テキストは「チェックボックスの値の表示」
  クリックした時は
    「チェックボックスの値は {甲の値} です。」と言う
```

10.2.4　ラジオボタン

チェックの有無を表示するいくつかの選択肢から構成されていて、チェックの入った選択肢をクリックで変更することによって、それらの選択肢からひとつを選択することのできる GUI 部品は、「ラジオボタン」と呼ばれます。ラジオボタンは、「ラジオ」というグループ型から作ることができます。

ラジオボタンを構成する選択肢は、「アイテム」というメンバー変数に代入されたテキストから作られます。このメンバー変数には、選択肢として表示したいテキストのそれぞれを改行で区切って並べたものを代入します。

ラジオボタンを構成している選択肢のうちのどれが選択されているのかということは、「値」というメンバー変数の内容を調べることによって判定することができます。「値」の内容は、選択されている選択肢の番号です。選択肢には、上から順番に、0、1、2、3、……という番号が与えられています。

あらかじめ、ラジオボタンの特定の選択肢が選択されている状態にしておきたい場合は、メンバー変数の「値」に、その選択肢の番号を代入しておきます。

ラジオボタンが持っている「幅」と「高さ」というメンバー変数のそれぞれには、選択肢を表示するのに十分な長さが自動的に代入されます。

プログラムの例　radio.nako

```
甲とはラジオ
甲について
  アイテムは「第一の選択肢
第二の選択肢
第三の選択肢
第四の選択肢」
  値は 0
  テキストは「私はラジオボタンです。」
乙とはボタン
乙について
  幅は 200
  テキストは「ラジオボタンの値の表示」
  クリックした時は
    「ラジオボタンの値は {甲の値} です。」と言う
```

10.3 テキストに関連する GUI 部品

10.3.1 この節について

GUI 部品の中には、テキスト（文字列）を表示したりテキストを入力または編集したりすることを目的とするものがあります。この節では、そのような GUI 部品を紹介したいと思います。

10.3.2 ラベル

テキストを表示するという機能のみを持つ GUI 部品は、「ラベル」と呼ばれます。ラベルは、「ラベル」というグループ型から作ることができます。

ラベルを使って表示したいテキストは、「テキスト」というメンバー変数に代入します。改行を含んでいるテキストを表示することも可能です。

ラベルが持っている「幅」と「高さ」というメンバー変数のそれぞれには、テキストを表示するのに十分な長さが自動的に代入されます。

プログラムの例 `label.nako`

```
甲とはラベル
甲について
  テキストは「私はラベルです。」
乙とはラベル
乙について
  テキストは「ラベルは、
改行を含んでいるテキストも
表示することができます。」
```

10.3.3 テキストフィールド

1 行のテキストを読み込むことのできる GUI 部品は、「テキストフィールド」、「テキストボックス」、「エディットボックス」などと呼ばれます。テキストフィールドは、「エディタ」というグループ型から作ることができます。

テキストフィールドに入力されたテキストは、「テキスト」というメンバー変数に代入されます。この変数にテキストを代入すると、そのテキストが入力された状態になります。

テキストフィールドが持っている「高さ」というメンバー変数には、適切な縦の長さが自動的に設定されます。「幅」というメンバー変数にも初期値が代入されますが、それが適切ではない場合は変更する必要があります。

画面上にあるいくつかの GUI 部品のうちで、キーボードからの入力が可能になっているものは、それに「フォーカス」があると言われます。テキストフィールドが持っている「注目」というメンバー命令を実行すると、そのテキストフィールドにフォーカスが移動します。

プログラムの例 `editor.nako`

```
甲とはエディタ
甲について
  幅は 200
  注目
乙とはボタン
乙について
  幅は 200
  テキストは「テキストの取得」
  クリックした時は
    「入力されたテキストは {甲のテキスト} です。」と言う
```

10.3.4 テキストエリア

複数行のテキストを読み込むことのできる GUI 部品は、「テキストエリア」と呼ばれます。テキストエリアは、「メモ」というグループ型から作ることができます。

テキストフィールドの場合と同じように、テキストエリアに入力されたテキストも、「テキスト」というメンバー変数に代入されます。この変数にテキストを代入すると、そのテキストが入力された状態になります。

テキストエリアが持っている「幅」と「高さ」というメンバー変数のそれぞれには初期値が代入されますが、それが適切ではない場合は、変更する必要があります。

テキストフィールドの場合と同じように、テキストエリアにフォーカスを移動させたい場合も、「注目」というメンバー命令を実行します。

プログラムの例　memo.nako

```
甲とはメモ
甲について
  幅は 300
  高さは 300
  注目
乙とはボタン
乙について
  位置は甲の右側
  幅は 200
  テキストは「テキストの取得」
  クリックした時は
    「入力されたテキストは {甲のテキスト} です。」と言う
```

10.4　リストとコンボボックス

10.4.1　リストとコンボボックスの基礎

ラジオボタンは、いくつかの選択肢の中からひとつをユーザーに選択してもらうために使われる GUI 部品ですが、そのような目的で使うことのできる GUI 部品は、ラジオボタンだけではありません。「リスト」と呼ばれる GUI 部品や、「コンボボックス」と呼ばれる GUI 部品も、ラジオボタンと同じ目的で使うことができます。

10.4.2　リスト

リストは、ラジオボタンと同じように、いくつかの選択肢を表示して、クリックされた選択肢を選択された状態にする、という GUI 部品です。リストは、「リスト」というグループ型から作ることができます。

リストを構成する選択肢は、「アイテム」というメンバー変数に代入されたテキストから作られます。このメンバー変数には、選択肢として表示したいテキストのそれぞれを改行で区切って並べたものを代入します。

リストを構成している選択肢のうちのどれが選択されているのかということは、「値」または「テキスト」というメンバー変数の内容を調べることによって判定することができます。

「値」の内容は、選択されている選択肢の番号です。選択肢には、上から順番に、0、1、2、3、……という番号が与えられています。

「テキスト」の内容は、選択されている選択肢のテキストです。

あらかじめ、リストの特定の選択肢が選択されている状態にしておきたい場合は、メンバー変数の「値」に、その選択肢の番号を代入しておきます。

プログラムの例　list.nako

```
甲とはリスト
甲について
  アイテムは「第一の選択肢
第二の選択肢
第三の選択肢
第四の選択肢」
  値は 2
乙とはボタン
乙について
  幅は 240
  テキストは「リストの値とテキストの表示」
  クリックした時は
    「値は {甲の値} でテキストは {甲のテキスト} です。」と言う
```

10.4.3　コンボボックス

「コンボボックス」は、リストとテキストフィールドとを組み合わせた GUI 部品です。この GUI 部品を使うと、ユーザーは、選択したいものが選択肢の中にある場合はそれを選択することができて、選択肢の中にない場合は任意のテキストを入力することができます。

コンボボックスは、「コンボ」というグループ型から作ることができます。

コンボボックスを構成する選択肢の作り方や、選択されている選択肢を判定する方法は、リストの場合と同じです。選択肢が選択されるのではなく、テキストが入力された場合は、「値」には -1 が代入されて、「テキスト」には入力されたテキストが代入されます。

プログラムの例　combo.nako

```
甲とはコンボ
甲について
  アイテムは「第一の選択肢
第二の選択肢
第三の選択肢
第四の選択肢」
  値は 2
乙とはボタン
乙について
  幅は 280
  テキストは「コンボボックスの値とテキストの表示」
  クリックした時は
    「値は {甲の値} でテキストは {甲のテキスト} です。」と言う
```

10.5 メニュー

10.5.1 メニューの基礎

ウィンドウのタイトルバーの下に配置される、メニューを表示するための GUI 部品は、「メニューバー」と呼ばれます。メニューバーは、「メインメニュー」というグループ型から作ることができます。

メニューバーの上には、メニューを何個でも表示することができます。メニューは、「メニュー」というグループ型から作られる GUI 部品です。

メニューが持っている「テキスト」というメンバー変数にテキストを代入すると、そのテキストがメニューの上に表示されます。

メニューがクリックされたときに実行されるイベント処理は、「クリックした時」というイベントを使って定義します。

10.5.2 メニューの構築

メニューバーとメニューは、その中にメニューを何個でも持つことのできる容器になっています。ですから、メニューは、何重にでも入れ子にすることができます。外側のメニューと内側のメニューは、上位と下位の関係になります。

メニューバーまたはメニューの中にメニューを入れたいときは、「追加」という命令を使います。「追加」を実行する文は、

$\boxed{\text{文}_1}$ を $\boxed{\text{文}_2}$ に追加

と書きます。「文 $_1$」のところには下位のメニューを求める文を書いて、「文 $_2$」のところには上位のメニューバーまたはメニューを求める文を書きます。

プログラムの例　menu.nako

```
甲とはメインメニュー
甲甲とはメニュー
甲甲のテキストは「料理」
甲甲甲とはメニュー
甲甲甲について
  テキストは「カレー」
  クリックした時は〜「カレー」を注文
甲甲乙とはメニュー
甲甲乙のテキストは「定食」
甲甲乙甲とはメニュー
甲甲乙甲について
  テキストは「とんかつ定食」
  クリックした時は〜「とんかつ定食」を注文
甲甲乙乙とはメニュー
```

```
甲甲乙乙について
  テキストは「焼肉定食」
  クリックした時は〜「焼肉定食」を注文
甲甲を甲に追加
甲甲甲を甲甲に追加
甲甲乙を甲甲に追加
甲甲乙甲を甲甲乙に追加
甲甲乙乙を甲甲乙に追加

●注文（甲を）
  「{甲} が注文されました。」と言う
```

10.5.3　ニーモニック

メニューがクリックされるというイベントは、マウスだけではなくて、キーボードで発生させることも可能です。

あらかじめ、特定の英字または数字のキーをメニューに割り当てておくと、Alt キーを押しながらそのキーを押したときに、そのメニューがクリックされるというイベントが発生します。そのような、Alt キーとともに押されたときにクリックのイベントを発生させるキーは、「ニーモニック」と呼ばれます。

メニューに対してニーモニックを割り当てたいときは、そのメニューが持っている「テキスト」というメンバー変数に、

& 英字または数字

という 2 文字を含むテキストを代入します。アンパサンド (&) とそれに続く文字は、どちらも半角でないといけません。英字は、大文字でも小文字でもどちらでもかまいません。たとえば、

焼きそば (&Y)

というテキストを「テキスト」に代入することによって、Y という文字のキーをニーモニックとして割り当てることができます。

プログラムの例　mnemonic.nako

```
甲とはメインメニュー
甲甲とはメニュー
甲甲のテキストは「料理 (&F)」
甲甲甲とはメニュー
甲甲甲について
  テキストは「中華丼 (&C)」
  クリックした時は〜「中華丼」を注文
甲甲乙とはメニュー
甲甲乙について
  テキストは「焼きそば (&Y)」
  クリックした時は〜「焼きそば」を注文
甲甲を甲に追加
甲甲甲を甲甲に追加
甲甲乙を甲甲に追加

●注文（甲を）
  「{甲} が注文されました。」と言う
```

10.6　ウィンドウ

10.6.1　ウィンドウの基礎

ウィンドウは、「フォーム」というグループ型から作ることのできる GUI 部品です。

第 9.1 節で、母艦のタイトルを変更する方法とクライアント領域の大きさを変更する方法について説明しましたが、そのときの説明は、母艦以外のウィンドウについても有効です。タイトルを変更したいときは「タイトル」というメンバー変数にタイトルを代入すればよくて、クライアント領域の大きさを変更したいときは「クライアント幅」と「クライアント高さ」というメンバー変数に横と縦の長さを代入すればいいのです。

ウィンドウは、表示または非表示という状態を持っています。母艦は、あらかじめ表示の状態

に設定されていますが、プログラムの中で宣言されたウィンドウは、デフォルトでは非表示の状態に設定されています。ウィンドウの状態は、「表示」というメンバー命令を実行することによって、非表示から表示へ変更することができます。

プログラムの例 `form.nako`

```
甲とはフォーム
甲について
  クライアント幅は 300
  クライアント高さは 200
  タイトルは「私はウィンドウです。」
  表示
```

10.6.2 ウィンドウへの GUI 部品の配置

プログラムの中で GUI 部品を宣言すると、その GUI 部品は、デフォルトでは母艦の上に配置されます。つまり、デフォルトでは母艦がその GUI 部品の親になっているということです。

母艦の上ではなくて、母艦以外のウィンドウの上に GUI 部品を配置したいときは、その GUI 部品の親を変更する必要があります。

すべての GUI 部品は、「親部品」というメンバー変数を持っていて、そこには、その GUI 部品の親が代入されています。このメンバー変数に別の GUI 部品を代入することによって親を変更すると、その GUI 部品は、新しい親の上に配置されることになります。

プログラムの例 `oyabuhin.nako`

```
甲とはフォーム
甲について
  タイトルは「第二ウィンドウ」
  表示
乙とはボタン
乙について
  幅は 300
  テキストは「私は第二ウィンドウのボタンです。」
  親部品は甲
```

10.6.3 ウィンドウを閉じる手段の提供

ウィンドウを閉じるというのは、ウィンドウの状態を表示から非表示へ変更するということです。

ウィンドウは、通常、タイトルバーにある×印のボタンをクリックすることによって閉じることができるわけですが、場合によっては、ウィンドウを閉じるための手段をそれ以外にも提供したいことがあります。

ウィンドウは、「閉じる」というメンバー命令を持っています。これは、ウィンドウを閉じるという動作、つまりウィンドウの状態を表示から非表示へ変更するという動作を実行する命令です。この命令を使えば、ウィンドウを閉じる手段として、×印のボタン以外のものを提供することができます。

次のプログラムが表示するウィンドウは、クライアント領域をクリックすることによって閉じることができます。

プログラムの例 `tojiru.nako`

```
甲とはフォーム
甲について
  タイトルは「私はクリックで閉じることもできます。」
  クリックした時は〜閉じる
  表示
```

閉じたウィンドウ、つまり非表示の状態にあるウィンドウは、再び「表示」を実行することによって表示の状態に戻すことが可能です。ただし、母艦に関しては、それを閉じた時点でプログラムの実行が終了しますので、閉じたのちに表示の状態に戻すことはできません。

10.6.4 オリジナルなダイアログボックス

第 9.2 節で説明したように、定型的なダイアログボックスは、命令を実行するだけで表示することができるわけですが、定型的ではないオリジナルなダイアログボックスを表示するためには、

GUI 部品としてそれを作る必要があります。
　ダイアログボックスというのはウィンドウの一種ですので、ダイアログボックスも、普通のウィンドウと同じように、「フォーム」というグループ型から作られます。
　ウィンドウは、デフォルトでは、ユーザーによる大きさの変更、最大化、最小化が可能な状態になっています。しかし、ウィンドウをダイアログボックスとして使う場合は、それらの操作ができない状態にするのが普通です。
　ウィンドウは、「スタイル」というメンバー変数を持っています。この変数に「ダイアログスタイル」という文字列を代入すると、そのウィンドウは、ユーザーによる大きさの変更、最大化、最小化ができない状態になります。

プログラムの例　dialogbox.nako

```
甲とはフォーム
甲について
  幅は 300
  高さは 200
  スタイルは「ダイアログスタイル」
  タイトルは「私はダイアログボックスです。」
乙とはボタン
乙について
  幅は 240
  テキストは「ダイアログボックスの表示」
  クリックした時は〜甲の表示
```

10.6.5　モーダルなウィンドウ

　ウィンドウには、非モーダルなものと、モーダルなものとがあります。「非モーダル」なウィンドウというのは、それが表示されているあいだも、それを表示したウィンドウ（つまり親）に対する操作が可能であるようなウィンドウのことです。それに対して、「モーダル」なウィンドウというのは、それが表示されているあいだは、それを表示したウィンドウに対する操作が不可能になるようなウィンドウのことです。ダイアログボックスは、多くの場合、モーダルなウィンドウとして表示する必要があります。
　ウィンドウの状態を非表示から表示へ変更するメンバー命令としては、「表示」のほかに、「モーダル表示」というものもあります。「表示」によって表示されたウィンドウが非モーダルなウィンドウになるのに対して、「モーダル表示」によって表示されたウィンドウは、モーダルなウィンドウになります。

プログラムの例　modal.nako

```
甲とはフォーム
甲について
  幅は 300
  高さは 200
  スタイルは「ダイアログスタイル」
  タイトルは「私はモーダルなダイアログボックスです。」
乙とはボタン
乙について
  幅は 280
  テキストは「モーダルなダイアログボックスの表示」
  クリックした時は〜甲のモーダル表示
```

第 11 章　グラフィックス

11.1　グラフィックスの基礎

11.1.1　グラフィックスの座標系

　ウィンドウの上には、グラフィックスを描画することができます。
　グラフィックスの位置を指定するために使われるウィンドウの座標系は、第 10.1.1 項で説明した、GUI 部品を配置するための座標系と同じです。つまり、クライアント領域の左上の隅に原点があって、x 座標は右向き、y 座標は下向き、という座標系です。

11.1.2 長方形の描画

グラフィックスは、命令を実行することによって描画することができます。たとえば、「四角」という命令を実行することによって、ウィンドウの上に長方形を描画することができます。

「四角」を実行する文は、

四角（x_1、y_1、x_2、y_2）

と書きます。引数として渡すのは、長方形の左上の頂点の座標 (x_1, y_1) と右下の頂点の座標 (x_2, y_2) です。

次のプログラムは、$(100, 50)$ を左上の頂点の座標、$(450, 300)$ を右下の頂点の座標とする長方形を描画します。

プログラムの例 shikaku.nako

```
四角（100、50、450、300）
```

11.1.3 母艦以外のウィンドウへの描画

グラフィックスを描画する命令は、デフォルトでは、母艦の上にグラフィックスを描画します。母艦以外のウィンドウの上にグラフィックスを描画したいときは、グラフィックスを描画する命令の先頭に、

ウィンドウのグループ名 の

と書くことによって、そのウィンドウを指定します。たとえば、

甲の四角（100、50、450、300）

と書くことによって、「甲」というグループ名を持つウィンドウの上に長方形を描画することができます。

プログラムの例 betsuwindowshikaku.nako

```
甲とはフォーム
甲について
  タイトルは「第二ウィンドウ」
  表示
甲の四角（100、50、450、300）
```

11.1.4 イメージラベル

グラフィックスをその上に描画することのできる GUI 部品は、ウィンドウだけではありません。グラフィックスは、「イメージラベル」と呼ばれる GUI 部品の上にも描画することができます。イメージラベルは、「イメージ」というグループ型から作ることができます。

イメージラベルの座標系は、左上の隅に原点があって、x 座標は右向き、y 座標は下向きです。

イメージラベルの上にグラフィックスを描画したいときは、グラフィックスを描画する命令の先頭に、

イメージラベルのグループ名 の

と書くことによって、そのイメージラベルを指定します（グループ名は、について文を使うことによって省略することも可能です）。たとえば、

甲の四角（0、0、300、200）

と書くことによって、「甲」というグループ名を持つイメージラベルの上に長方形を描画することができます。

プログラムの例 image.nako

```
甲とはイメージ
甲について
  位置は「150,50」
  幅は 300
  高さは 200
  四角（0、0、300、200）
```

11.2　描画属性

11.2.1　描画属性の基礎

グラフィックスを描画するときに使われる各種の属性は、「描画属性」と呼ばれます。

なでしこでは、描画属性を保持するための各種の変数が、あらかじめ宣言されています。それらの変数に別の属性を代入すると、それ以降は、代入された属性でグラフィックスが描画されることになります。

11.2.2　塗りつぶしのスタイル

グラフィックスの内部をどのようなスタイルで塗りつぶすのかという描画属性は、「塗りスタイル」という変数によって保持されています。この変数には、

べた　透明　格子　十字線　縦線　横線　右斜め線　左斜め線　斜め十字線

という文字列のいずれかを代入することができます（「格子」と「十字線」は同じ意味）。デフォルトは「べた」です。

プログラムの例　nuristyle.nako

```
塗りスタイルは「格子」
四角（50、30、350、250）
塗りスタイルは「左斜め線」
四角（120、80、420、300）
塗りスタイルは「透明」
四角（190、130、490、350）
```

11.2.3　線のスタイル

グラフィックスの線をどのようなスタイルで描画するのかという描画属性は、「線スタイル」という変数によって保持されています。この変数には、

実線　点線　破線　透明

という文字列のいずれかを代入することができます。デフォルトは「実線」です。

プログラムの例　senstyle.nako

```
塗りスタイルは「透明」
線スタイルは「点線」
四角（50、30、350、250）
線スタイルは「破線」
四角（120、80、420、300）
塗りスタイルは「右斜め線」
線スタイルは「透明」
四角（190、130、490、350）
```

11.2.4　線の太さ

描画するグラフィックスの線の太さという描画属性は、「線太さ」という変数によって保持されています。線の太さは、数値（単位はピクセル）によって示されます。デフォルトは 3 ピクセルです。

「線太さ」に代入されている太さで線を描画するためには、「線スタイル」に「実線」が代入されている必要があります。

プログラムの例　senfutosa.nako

```
塗りスタイルは「透明」
線太さは 5
四角（50、30、350、250）
線太さは 10
四角（120、80、420、300）
線太さは 20
四角（190、130、490、350）
```

11.2.5 色をあらわす整数

なでしこでは、グラフィックスの色は 1 個の整数によってあらわされます。たとえば、黒という色は 0 という整数によってあらわされ、赤という色は 16711680 という整数によってあらわされます。

色をあらわす整数は、光の三原色を構成している、赤、緑、青というそれぞれの原色の明るさを示しています。それぞれの原色の明るさは、0 から 255 までの整数によってあらわされます。2 桁の 16 進数で表記すると、00 から ff までです。

赤の明るさを示す整数を 2 桁の 16 進数で表記したものを *rr*、緑の明るさを示す整数を 2 桁の 16 進数で表記したものを *gg*、青の明るさを示す整数を 2 桁の 16 進数で表記したものを *bb* とすると、それらを混ぜ合わせることによってできる色は、

rrggbb

という 6 桁の 16 進数で表記される整数によってあらわされます。たとえば、赤という色は、

ff0000

という 6 桁の 16 進数で表記される整数によってあらわされます。これを 10 進数で表記すると、16711680 になります。

なでしこでは、このような方法を使って色を 1 個の整数であらわしますので、プログラムの中で色を記述するときは、通常、16 進数リテラルが使われます。「16 進数リテラル」というのは、数値リテラルの一種で、16 進数で整数を記述しているリテラルのことです。

16 進数リテラルは、

`$` 16 進数

と書きます。たとえば、16 進数の ff0000 は、

```
$ff0000
```

という 16 進数リテラルによって記述されます。

11.2.6 塗りつぶしの色

グラフィックスの内部をどのような色で塗りつぶすのかという描画属性は、「塗り色」という変数によって保持されています。デフォルトは、黒をあらわす 0 です。

プログラムの例 nuriiro.nako

```
線スタイルは「透明」
塗り色は$ffa500　＃　オレンジ色
四角（50、30、350、250）
塗り色は$ffff00　＃　黄色
四角（120、80、420、300）
塗り色は$800000　＃　栗色
四角（190、130、490、350）
```

11.2.7 線の色

グラフィックスの線をどのような色で描画するのかという描画属性は、「線色」という変数によって保持されています。デフォルトは、黒をあらわす 0 です。

プログラムの例 seniro.nako

```
塗りスタイルは「透明」
線太さは 30
線色は$87ceeb　＃　空色
四角（50、30、350、250）
線色は$00ffff　＃　水色
四角（120、80、420、300）
線色は$000080　＃　ネイビー
四角（190、130、490、350）
```

11.2.8 色を選択するダイアログボックス

定型的なダイアログボックスを表示する命令としては、これまでに、「言う」や「尋ねる」や「二択」などを紹介していますが、ここで、さらにもうひとつ、定型的なダイアログボックスを

表示する命令を紹介しましょう。それは、「色選択」という命令です。

「色選択」は、ユーザーに色を選択してもらうためのダイアログボックスを表示します。引数は何も受け取りません。ユーザーが「OK」のボタンでダイアログボックスを閉じた場合は、そのときに選択されていた色をあらわす整数が戻り値になります。「キャンセル」のボタンで閉じた場合は、黒をあらわす整数が戻り値になります。

次のプログラムは、ダイアログボックスを使って、長方形を塗りつぶす色を選択することができます。

プログラムの例　irosentaku.nako

```
甲とはボタン
甲について
  幅は 100
  テキストは「色の変更」
  クリックした時は
    塗り色は色選択
    長方形
線スタイルは「透明」
長方形

●長方形
  四角（10、50、400、300）
```

11.3　描画の命令

11.3.1　この節について

第 11.1.2 項で説明したように、「四角」という命令を実行することによって、長方形を描画することができます。

グラフィックスを描画する命令は、「四角」だけではありません。そこでこの節では、グラフィックスを描画する、それ以外の命令を紹介したいと思います。

11.3.2　直線

「線」という命令は、直線を描画します。

「線」を実行する文は、

線（x_1、y_1、x_2、y_2）

と書きます。引数として渡すのは、(x_1, y_1) と (x_2, y_2) という 2 個の点の座標です。「線」は、引数で指定された 2 個の点をつなぐ直線を描画します。

次のプログラムは、$(100, 50)$ と $(450, 300)$ とをつなぐ直線を描画します。

プログラムの例　sen.nako

```
線（100、50、450、300）
```

11.3.3　楕円

「円」という命令は、楕円を描画します。

「円」を実行する文は、

円（x_1、y_1、x_2、y_2）

と書きます。引数として渡すのは、長方形の左上の頂点の座標 (x_1, y_1) と右下の頂点の座標 (x_2, y_2) です。「円」は、引数で指定された長方形に内接する楕円を描画します。

次のプログラムは、$(100, 50)$ を左上の頂点の座標、$(450, 300)$ を右下の頂点の座標とする長方形に内接する楕円を描画します。

プログラムの例　en.nako

```
円（100、50、450、300）
```

11.3.4　角の丸い長方形

「角丸四角」という命令は、角の丸い長方形を描画します。

「角丸四角」を実行する文は、

角丸四角（x_1、y_1、x_2、y_2、d_x、d_y）

と書きます。引数として渡すのは、長方形の左上の頂点の座標(x_1, y_1)、右下の頂点の座標(x_2, y_2)、そして、楕円のx軸方向の直径d_xとy軸方向の直径d_yです。「角丸四角」は、引数で指定された長方形を描画して、指定された楕円に沿って、その角を丸くします。

次のプログラムは、$(100, 50)$を左上の頂点の座標、$(450, 300)$を右下の頂点の座標とする長方形を描画して、175をx軸方向の直径、125をy軸方向の直径とする楕円に沿って、その角を丸くします。

プログラムの例 kadomarushikaku.nako

```
角丸四角（100、50、450、300、175、125）
```

11.3.5 多角形

「多角形」という命令は、多角形を描画します。

「多角形」は、座標をコンマで区切って並べた、

x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4,$\cdots$

という形の文字列を引数として受け取って（助詞は「で」）、それぞれの座標が示している点を、その順番のとおりに直線でつないでいって、さらに最後の点と最初の点も直線でつなぐことによってできる多角形を描画します。たとえば、

```
150,50,200,300,100,300
```

という文字列を引数として渡して「多角形」を実行することによって、$(150, 50)$と$(200, 300)$と$(100, 300)$と$(150, 50)$とを直線でつなぐことによってできる三角形を描画することができます。

次のプログラムは、星形五角形を描画します。

プログラムの例 takakkei.nako

```
塗りスタイルは「透明」
「100,140,400,140,130,300,250,50,370,300」で多角形
```

11.3.6 グラフィックスの消去

「画面クリア」という命令は、GUI部品を塗りつぶすことによって、描画されているグラフィックスを消去します。

「画面クリア」は、色をあらわす整数を引数として受け取って（助詞は「で」）、その色でGUI部品を塗りつぶします。引数を省略すると、白色で塗りつぶします。

次のプログラムは、イメージラベルがマウスでクリックされると、その位置に円を描画します。描画した円は、「白色で消去する」というボタンをクリックすると白色で消去されて、「黄緑色で消去する」というボタンをクリックすると黄緑色で消去されます。

プログラムの例 gamenclear.nako

```
線スタイルは「透明」
塗り色は$000080
白とはボタン
白について
  幅は 140
  テキストは「白色で消去する」
  クリックした時は〜キャンバスの画面クリア
黄緑とはボタン
黄緑について
  幅は 140
  テキストは「黄緑色で消去する」
  クリックした時は〜キャンバスの$99ff33で画面クリア
キャンバスとはイメージ
キャンバスについて
  幅は 400
  高さは 300
  マウス離した時は
    R とは整数 ＝ 20
    X とは整数 ＝ マウス X
```

```
    Y とは整数　＝　マウス Y
    円（X から R を引く、Y から R を引く、X に R を足す、Y に R を足す）
```

11.4　文字列の描画

11.4.1　文字列を描画する命令

この節では、文字列（テキスト）をグラフィックスとして描画する方法について説明したいと思います。

文字列をグラフィックスとして描画したいときは、「文字描画」という命令を使います。

「文字描画」を実行する文は、

文字描画（x、y、文字列）

と書きます。引数として渡すのは、文字列の左上の位置を指定する座標 (x, y) と、描画する文字列です。

次のプログラムは、「私は文字列です。」という文字列を $(100, 50)$ という位置に描画します。

プログラムの例　mojibyouga.nako

```
文字描画（100、50、「私は文字列です。」）
```

11.4.2　文字の大きさ

文字列を構成しているそれぞれの文字をどのような大きさで描画するのか、という描画属性は、「文字サイズ」という変数によって保持されています。文字の大きさは、数値（単位はピクセル）によって示されます。デフォルトは 10 ピクセルです。

プログラムの例　mojisize.nako

```
文字サイズは 20
文字描画（50、50、「私の大きさは 20 です。」）
文字サイズは 30
文字描画（50、100、「私の大きさは 30 です。」）
文字サイズは 40
文字描画（50、170、「私の大きさは 40 です。」）
```

11.4.3　文字の色

文字列を描画する色という描画属性は、「文字色」という変数によって保持されています。デフォルトは、黒をあらわす 0 です。

プログラムの例　mojiiro.nako

```
文字サイズは 30
文字色は$7fff00
文字描画（50、50、「私の色はシャトルーズです。」）
文字色は$dc143c
文字描画（50、120、「私の色はクリムゾンです。」）
文字色は$4b0082
文字描画（50、190、「私の色はインディゴです。」）
```

11.4.4　文字列の配置

テキストを描画するために指定した位置と、実際に描画されるテキストとの水平方向の位置関係は、そのテキストの「配置」(alignment) と呼ばれます。

指定された位置をテキストの左端にする配置は「左寄せ」、指定された位置をテキストの右端にする配置は「右寄せ」、指定された位置をテキストの中央にする配置は「センタリング」と呼ばれます。

「文字描画」という命令は、引数で指定された位置をテキストの左端にしますので、この命令を使ってテキストを描画すると、テキストは自動的に左寄せになります。テキストを右寄せにしたりセンタリングしたりするためには、描画される文字列の水平方向の長さを求める必要があります。

「文字幅取得」という命令は、引数として文字列を受け取って（助詞は「の」）、その文字列を描画したときの水平方向の長さ（単位はピクセル）を戻り値として返します。ですから、この命令を使うことによって、文字列を描画するときに、それを右寄せにしたり、センタリングしたりすることができます。

プログラムの例　mojiretsuhaichi.nako

```
文字サイズは 40
X とは整数　＝　300
甲とは文字列　＝　「左寄せ」
乙とは文字列　＝　「右寄せ」
丙とは文字列　＝　「センタリング」
線（X、30、X、310）
文字描画（X、50、甲）
文字描画（X から乙の文字幅取得を引く、140、乙）
文字描画（X から丙の文字幅取得を 2 で割るを引く、230、丙）
```

11.5　画像

11.5.1　画像の描画

この節では、画像に関する処理について説明したいと思います。

ウィンドウまたはイメージラベルの上には、画像を描画することも可能です。画像を描画したいときは、「画像描画」という命令を使います。

「画像描画」を実行する文は、

画像描画（x、y、 パス名 ）

と書きます。引数として渡すのは、画像の左上の位置を指定する座標 (x, y) と、画像が保存されているファイルのパス名です。

次のプログラムは、プログラムのファイルと同じフォルダに置かれている「sample.jpg」というファイルに保存されている画像を $(100, 50)$ という位置に描画します。

プログラムの例　gazoubyouga.nako

```
画像描画（100、50、「sample.jpg」）
```

11.5.2　画像の保存

GUI 部品の上に描画したグラフィックスは、画像としてファイルに保存することができます。保存が可能な画像のデータ形式は、JPEG、PNG、GIF などです。

グラフィックスを画像としてファイルに保存したいときは、「画像保存」という命令を使います。

「画像保存」を実行する文は、

GUI 部品 を パス名 に画像保存

と書きます。引数として渡すのは、グラフィックスが描画されている GUI 部品と、そのグラフィックスを保存するファイルのパス名です。画像のデータ形式は、ファイル名の拡張子によって決まります。たとえば、 .jpg ならば JPEG、 .png ならば PNG になります。

次のプログラムは、イメージラベルがマウスでクリックされると、その位置に円を描画します。「画像を保存する」というボタンをクリックすると、ウィンドウに描画されているグラフィックスを、 gazou.jpg というファイルに保存します。

プログラムの例　gazouhozon.nako

```
線スタイルは「透明」
塗り色は$008000
甲とはボタン
甲について
  幅は 140
  テキストは「画像を保存する」
  クリックした時は〜キャンバスを「gazou.jpg」に画像保存
キャンバスとはイメージ
キャンバスについて
  幅は 400
```

```
  高さは 300
  マウス離した時は
    R とは整数　＝　 20
    X とは整数　＝　マウス X
    Y とは整数　＝　マウス Y
    円（X から R を引く、Y から R を引く、X に R を足す、Y に R を足す）
```

11.5.3　クリップボード

クリップボードにある画像は、GUI 部品の上にペーストすることが可能です。逆に、GUI 部品の上に描画されたグラフィックスを画像としてクリップボードにコピーすることも可能です。

「画像描画」という命令は、3 個目の引数としてパス名を受け取るわけですが、パス名の代わりに、「クリップボード」という文字列を引数として渡すと、クリップボードにある画像を GUI 部品にペーストします。

同じように、「画像保存」という命令も、パス名ではなく、「クリップボード」という文字列を引数として渡すと、GUI 部品からクリップボードへ画像をコピーします。

次のプログラムは、イメージラベルがマウスでクリックされると、その位置に円を描画します。「コピー」というボタンをクリックすると、イメージラベルに描画されているグラフィックスをクリップボードにコピーします。「ペースト」というボタンをクリックすると、クリップボードにある画像をイメージラベルに描画します。

プログラムの例　clipboard.nako

```
線スタイルは「透明」
塗り色は$800000
甲とはボタン
甲について
  幅は 100
  テキストは「コピー」
  クリックした時は〜キャンバスを「クリップボード」に画像保存
乙とはボタン
乙について
  幅は 100
  テキストは「ペースト」
  クリックした時は
    キャンバスの画像描画（0、0、「クリップボード」）
キャンバスとはイメージ
キャンバスについて
  位置は甲の右側
  幅は 400
  高さは 300
  マウス離した時は
    R とは整数　＝　 20
    X とは整数　＝　マウス X
    Y とは整数　＝　マウス Y
    円（X から R を引く、Y から R を引く、X に R を足す、Y に R を足す）
```

11.6　アニメーション

11.6.1　アニメーションの基礎

時間の経過とともに変化するグラフィックスは、「アニメーション」と呼ばれます。

第 9.4 節で説明したように、「タイマー」というグループ型名の GUI 部品は、一定の時間間隔でイベントを発生させます。ですから、この GUI 部品を使うことによって、時間の経過とともにグラフィックスを変化させていくということ、つまりアニメーションを作るということができます。

11.6.2　定期的な描画

グラフィックスが移動するアニメーションは、タイマーを使って、位置を移動させながらグラフィックスを定期的に描画することによって作ることができます。

次のプログラムは、位置を右へ移動させながら、タイマーを使って正方形を定期的に描画します。ただし、過去に描画した正方形は、そのまま残像として存在し続けます。

プログラムの例 zanzou.nako

```
線スタイルは「透明」
塗り色は$ff1493
X とは整数 ＝ 0
甲とはタイマー
甲について
  値は 10
  時満ちた時は
    X は X に 1 を足す
    四角（X、140、X に 20 を足す、160）
  開始
```

11.6.3 残像の消去

アニメーションを作るためには、多くの場合、ただ単にグラフィックスを定期的に描画するだけではなくて、残像となったグラフィックスを消去することも必要になります。

残像となったグラフィックスは、「画面クリア」を使うことによって消去することができます。

次のプログラムは、先ほどのプログラムと同じように、位置を右へ移動させながら、タイマーを使って正方形を定期的に描画します。ただし、先ほどとは違って、描画する前に、残像となった正方形を消去します。

プログラムの例 zanzounoshoukyo.nako

```
線スタイルは「透明」
塗り色は$32cd32
X とは整数 ＝ 0
甲とはタイマー
甲について
  値は 10
  時満ちた時は
    X は X に 1 を足す
    画面クリア
    四角（X、140、X に 20 を足す、160）
  開始
```

11.6.4 インタラクティブなアニメーション

次に、インタラクティブなアニメーション、つまりユーザーが操作することのできるアニメーションを作ってみましょう。

次のプログラムは、矢印のキーを押すことによって、正方形が移動する方向を変更することができます。

プログラムの例 houkounohenkou.nako

```
線スタイルは「透明」
塗り色は$ff4500
X とは整数 ＝ 0
Y とは整数 ＝ 140
方向とは整数 ＝ 3 # 右
母艦のキー押した時は
  方向は母艦の押された仮想キーから 36 を引く
甲とはタイマー
甲について
  値は 10
  時満ちた時は
    画面クリア
    四角（X、Y、X に 20 を足す、Y に 20 を足す）
    移動
  開始

●移動
  方向で条件分岐
    1 ならば # 左
      X は X から 1 を引く
    2 ならば # 上
      Y は Y から 1 を引く
```

```
3ならば　#　右
  XはXに1を足す
4ならば　#　下
  YはYに1を足す
```

第 12 章　ファイル

12.1　ファイルを選択するダイアログボックス

12.1.1　この節について

なでしこでは、組み込み命令を実行することによって、定型的なダイアログボックスを表示することができます。これまでに、そのような組み込み命令として、「言う」や「尋ねる」や「二択」などを紹介しましたが、この節では、ファイルを選択するダイアログボックスを表示する組み込み命令を紹介したいと思います。

12.1.2　すでに存在するファイルを選択するダイアログボックス

すでに存在するファイルの中からユーザーにファイルを選択してもらいたいときは、「ファイル選択」という命令を使います。

「ファイル選択」を実行する文は、

　初期ファイル で 拡張子 のファイル選択

と書きます。引数として渡すのは、最初に選択されているファイルのパス名（ワイルドカードを使うことも可能）と、選択の対象とするファイルの拡張子です。引数として拡張子を渡さなかった場合、または拡張子として空文字列を渡した場合は、任意の拡張子のファイルが選択の対象になります。

「ファイル選択」は、ユーザーが「開く」というボタンでダイアログボックスを閉じた場合は、選択されたファイルのパス名を戻り値として返します。「キャンセル」というボタンで閉じた場合は、空文字列を返します。

プログラムの例　filesentaku.nako

```
「c:\Program Files\*.exe」で「.exe」のファイル選択を言う
```

12.1.3　データを保存するファイルを選択するダイアログボックス

「ファイル選択」を使って表示したダイアログボックスは、存在しないファイルを選択することができないようになっています。しかし、データを保存する場合には、すでに存在しているファイルだけではなくて、これから作るファイルの名前を入力することもできる必要があります。

データを保存するファイルを選択するためのダイアログボックスは、「保存ファイル選択」という命令を使うことによって、表示することができます。

「保存ファイル選択」を実行する文も、「ファイル選択」の場合と同じように、

　初期ファイル で 拡張子 の保存ファイル選択

と書きます。

「保存ファイル選択」は、すでに存在するファイルをユーザーが選択した場合、「上書きしますか？」とユーザーに尋ねます。そして、「はい」がクリックされた場合は、そのファイルのパス名を戻り値として返して終了しますが、「いいえ」がクリックされた場合は、ユーザーがファイルを選択することのできる状態に戻ります。

プログラムの例　hozonfilesentaku.nako

```
「c:\*.txt」で「.txt」の保存ファイル選択を言う
```

12.1.4　フォルダを選択するダイアログボックス

「ファイル選択」を使ってダイアログボックスを表示した場合、ファイルを選択することはできますが、フォルダを選択することはできません。ですから、ファイルではなくてフォルダをユーザーに選択してもらいたい場合、その命令は使えません。

フォルダを選択するためのダイアログボックスは、「フォルダ選択」という命令を使うことによって表示することができます。
「フォルダ選択」を実行する文は、

[初期フォルダ]でフォルダ選択

と書きます。引数として渡すのは、最初に選択されているフォルダのパス名です。
「フォルダ選択」は、ユーザーが「OK」というボタンでダイアログボックスを閉じた場合は、選択されたフォルダのパス名（末尾は常にバックスラッシュ[1]）を戻り値として返します。「キャンセル」というボタンで閉じた場合は、空文字列を返します。

プログラムの例 foldersentaku.nako

```
「c:\」でフォルダ選択を言う
```

12.2 読み込みと保存

12.2.1 ファイルからの読み込み

この節では、ファイルから文字列を読み込む方法と、ファイルに文字列を保存する方法について説明したいと思います。
ファイルから文字列を読み込みたいときは、「読む」という命令を使います。
「読む」を実行する文は、

[変数名]に[パス名]から読む

と書きます。「変数名」のところには、ファイルから読み込んだ文字列を代入する変数の名前を書きます。「パス名」のところには、そこから文字列を読み込みたいファイルのパス名を求める文を書きます。
「読む」は、パス名で指定されたファイルから文字列を読み込んで、その文字列を変数に代入します。
次のプログラムは、ファイルから文字列を読み込んで、その文字列をダイアログボックスで表示します。

プログラムの例 yomu.nako

```
パス名とは文字列 ＝ 「c:\*.*」でファイル選択
もしパス名が「」でないならば
  甲とは文字列
  甲にパス名から読む
  甲をメモ記入
```

12.2.2 ファイルへの保存

文字列をファイルに保存したいときは、「保存」という命令を使います。
「保存」を実行する文は、

[文字列]を[パス名]に保存

と書きます。引数として渡すのは、ファイルに保存したい文字列と、そこへ文字列を保存たいファイルのパス名です。
「保存」は、パス名で指定されたファイルに文字列を保存します。
次のプログラムは、ダイアログボックスで入力された文字列をファイルに保存します。

プログラムの例 hozon.nako

```
甲とは文字列 ＝ 「自由に編集してください。」をメモ記入
パス名とは文字列 ＝ 「c:\*.*」で保存ファイル選択
もしパス名が「」でないならば
  甲をパス名に保存
```

[1]日本語の環境では、通常、バックスラッシュ(\) は円マーク (¥) で表示されます。

12.2.3　ファイルの末尾への追加

「保存」という命令は、すでに存在しているファイルが指定された場合、そのファイルの内容を消去したのちに新しい文字列を保存します。ですから、以前の内容を消去するのではなくて、その末尾に文字列を追加したいという場合、「保存」を使うことはできません。

すでに存在するファイルが指定されたときは、そのファイルの末尾に文字列を追加したい、という場合は、「保存」ではなくて、「追加保存」という命令を使います。

「追加保存」を実行する文は、「追加」を実行する文と同じように、

　文字列 を パス名 に追加保存

と書きます。そうすると、パス名で指定されたファイルに文字列を保存します。ただし、「追加」とは違って、指定されたファイルがすでに存在する場合は、そのファイルの内容を消去しないで、その末尾に文字列を追加します。

次のプログラムは、ダイアログボックスで入力された文字列をファイルに保存します。指定されたファイルがすでに存在する場合は、その末尾に文字列を追加します。

プログラムの例　tsuikahozon.nako

```
甲とは文字列　＝　「自由に編集してください。」をメモ記入
パス名とは文字列　＝　「c:\*.*」で保存ファイル選択
もしパス名が「」でないならば
　甲をパス名に追加保存
```

12.3　ファイルストリーム

12.3.1　ファイルストリームの基礎

第 12.2.1 項で説明したように、ファイルから文字列を読み込みたいときは「読む」という命令を使えばいいわけですが、この命令は、ファイルに保存されている文字列の全体を読み込むことしかできません。ですから、ファイルに保存されている文字列を少しだけ読み込んでそれを処理するということを繰り返したいという場合には、この命令は使えません。

ファイルに保存されている文字列を少しだけ読み込むという処理を実行するためには、「ファイルストリーム」と呼ばれるものを使う必要があります。ファイルストリームというのは、プログラムとファイルとにあいだに作られる、その中をデータが流れる経路のことです。

12.3.2　ファイルストリームの準備

ファイルストリームを使うためには、使用に先立って、それを準備する必要があります。

ファイルストリームは、「ファイルストリーム開く」という命令を実行することによって準備することができます。

「ファイルストリーム開く」を実行する文は、

　パス名 を モード でファイルストリーム開く

と書きます。引数として渡すのは、準備するファイルストリームが読み書きの対象とするファイルのパス名と、ファイルに対する処理のモードを意味する文字列です。モードを意味する文字列は、読み込みならば「読」、書き込みならば「書」です。たとえば、

　「c:\hoge.txt」を「読」でファイルストリーム開く

という文を実行することによって、c:\hoge.txt というパス名で指定されたファイルから文字列を読み込むためのファイルストリームを準備することができます。

「ファイルストリーム開く」は、戻り値として、準備したファイルストリームを識別するための、「ハンドル」と呼ばれる整数を返します。ハンドルは、ファイルストリームを使ってファイルに対する読み書きを実行するとき、それを識別するために必要になります。

12.3.3　ファイルストリームの後片付け

ファイルストリームを使った読み書きが終了したときは、それの後片付けをする必要があります。

ファイルストリームは、「ファイルストリーム閉じる」という命令を実行することによって後片付けをすることができます。

「ファイルストリーム閉じる」を実行する文は、

ハンドル をファイルストリーム閉じる

と書きます。引数として渡すのは、後片付けをするファイルストリームのハンドルです。

12.3.4 バイト数を指定した読み込み

バイト数を指定してファイルから文字列を読み込みたいときは、「ファイルストリーム読む」という命令を使います。

「ファイルストリーム読む」を実行する文は、

ハンドル で バイト数 をファイルストリーム読む

と書きます。引数として渡すのは、使用するファイルストリームのハンドルと、読み込むバイト数です。

「ファイルストリーム読む」は、ハンドルで指定されたファイルストリームを使って、指定されたバイト数の文字列をファイルから読み込みます。戻り値は、読み込んだ文字列です。

次のプログラムは、ファイルから先頭の 100 バイトの文字列を読み込んで、その文字列をダイアログボックスで表示します。

プログラムの例 `filestreamyomu.nako`

```
パス名とは文字列 ＝ 「c:\*.*」でファイル選択
もしパス名が「」でないならば
  ハンドルとは整数 ＝ パス名を「読」でファイルストリーム開く
  ハンドルで 100 をファイルストリーム読むを言う
  ハンドルをファイルストリーム閉じる
```

12.3.5 ファイルの大きさ

ファイルの大きさ、つまり、そのファイルの中には何バイトのデータが保存されているかということは、「ファイルストリームサイズ」という命令を実行することによって調べることができます。

「ファイルストリームサイズ」を実行する文は、

ハンドル のファイルストリームサイズ

と書きます。「ファイルストリームサイズ」は、引数で指定されたファイルストリームの先にあるファイルの大きさを戻り値として返します。

次のプログラムは、ファイルの大きさをダイアログボックスで表示します。

プログラムの例 `filestreamsize.nako`

```
パス名とは文字列 ＝ 「c:\*.*」でファイル選択
もしパス名が「」でないならば
  ハンドルとは整数 ＝ パス名を「読」でファイルストリーム開く
  ハンドルのファイルストリームサイズを言う
  ハンドルをファイルストリーム閉じる
```

12.3.6 ファイルストリームの現在位置

ファイルストリームは、「現在位置」と呼ばれる、ファイルに対して文字を読み書きする位置を示している整数を保持しています。現在位置は、読み込みのモードでファイルストリームを開いた場合、初期値は 0 で、読み込みの命令を実行するたびに読み込んだバイト数だけ加算されます。ですから、読み込みの命令を実行したのち、さらに読み込みの命令を実行すると、前回の実行で読み込んだ部分の次の部分が読み込まれることになります。

現在位置は、「ファイルストリーム位置取得」という命令を実行することによって、ファイルストリームから取得することができます。

「ファイルストリーム位置取得」を実行する文は、

ハンドル のファイルストリーム位置取得

と書きます。「ファイルストリーム位置取得」は、引数で指定されたファイルストリームから現在位置を取得して、それを戻り値として返します。

ファイルの末尾の位置を示している整数は、ファイルの大きさと同じです。したがって、ファイルの現在位置がファイルの大きさに到達していないという条件で読み込みを繰り返すことによって、ファイルに格納されているすべてのデータを読み込むことができます。

次のプログラムは、ファイルに格納されている文字列を先頭から末尾まで 1 バイト単位で読み込んで、それらを連結した結果をダイアログボックスで表示します。

プログラムの例　filestreamichishutoku.nako

```
パス名とは文字列　＝　「c:\*.*」でファイル選択
もしパス名が「」でないならば
  ハンドルとは整数　＝　パス名を「読」でファイルストリーム開く
  甲とは文字列　＝　「」
  ハンドルが未完了の間
    甲は甲＆（ハンドルで 1 をファイルストリーム読む）
  ハンドルをファイルストリーム閉じる
  甲をメモ記入

●未完了（ハンドルが）
  サイズとは整数　＝　ハンドルのファイルストリームサイズ
  現在位置とは整数　＝　ハンドルのファイルストリーム位置取得
  現在位置が（サイズ）未満
```

12.3.7　行単位での読み込み

ファイルから行単位で文字列を読み込みたいときは、「ファイルストリーム一行読む」という命令を使います。

「ファイルストリーム一行読む」を実行する文は、

　[ハンドル]でファイルストリーム一行読む

と書きます。引数として渡すのは、使用するファイルストリームのハンドルです。

「ファイルストリーム一行読む」は、ハンドルで指定されたファイルストリームを使って、ファイルから一行の文字列を読み込みます。戻り値は、読み込んだ文字列です。

「ファイルストリーム一行読む」が戻り値として返す文字列は改行を含んでいませんので、改行が必要な場合は、それを追加する必要があります。なでしこは、「改行」という変数をあらかじめ宣言していて、そこには改行が代入されています。文字列に改行を追加するときには、この変数を使うと便利です。

次のプログラムは、ファイルに格納されている文字列を行単位で読み込んで、それぞれの行の先頭に行番号を追加したものをダイアログボックスで表示します。

プログラムの例　filestreamichigyouyomu.nako

```
パス名とは文字列　＝　「c:\*.*」でファイル選択
もしパス名が「」でないならば
  ハンドルとは整数　＝　パス名を「読」でファイルストリーム開く
  甲とは文字列　＝　「」
  行番号とは整数　＝　0
  行とは文字列
  ハンドルが未完了の間
    行番号は行番号に 1 を足す
    行はハンドルでファイルストリーム一行読む
    甲は甲＆行番号＆「: 」＆行＆改行
  ハンドルをファイルストリーム閉じる
  甲をメモ記入

●未完了（ハンドルが）
  サイズとは整数　＝　ハンドルのファイルストリームサイズ
  現在位置とは整数　＝　ハンドルのファイルストリーム位置取得
  現在位置が（サイズ）未満
```

12.3.8 毎行読んで反復文

ファイルから行単位で文字列を読み込む、ということをファイルの先頭から末尾まで繰り返す、という処理は、「毎行読む」という命令と、反復文」とを組み合わせることによって、簡単に記述することができます。

「毎行読む」を実行する文と反復文とを組み合わせた文のことを、便宜的に、「毎行読んで反復文」と呼ぶことにしましょう。毎行読んで反復文は、

パス名を毎行読んで反復
　プログラム

と書きます。この中の「パス名」のところには、読み込みの対象とするファイルのパス名を求める文を書きます。

毎行読んで反復文を実行すると、まず最初に、パス名で指定されたファイルから文字列を読み込むファイルストリームが準備されます。そして、ファイルから行単位で文字列を読み込んで、読み込んだ文字列を「それ」という変数に代入して、プログラムを実行する、ということがファイルの先頭から末尾まで繰り返されます。そして、その繰り返しが終了すると、ファイルストリームの後片付けが実行されます。

次のプログラムは、ファイルの内容に行番号を追加して表示する先ほどのプログラムを、毎行読んで反復文を使って書き直したものです。

プログラムの例　maigyouyondehanpukubun.nako

```
パス名とは文字列　＝　「c:\*.*」でファイル選択
もしパス名が「」でないならば
  甲とは文字列　＝　「」
  パス名を毎行読んで反復
    甲は甲＆回数＆「: 」＆それ＆改行
  甲をメモ記入
```

12.3.9 ファイルストリームを使った書き込み

ファイルストリームを使ってファイルに文字列を書き込みたいときは、「ファイルストリーム書く」という命令を使います。

「ファイルストリーム書く」を実行する文は、

ハンドルで文字列をファイルストリーム書く

と書きます。引数として渡すのは、使用するファイルストリームのハンドルと、読み込む文字列です。

「ファイルストリーム書く」は、ハンドルで指定されたファイルストリームを使って、文字列をファイルに書き込みます。

次のプログラムは、ファイルの内容に行番号を追加したものを、別のファイルに保存します。

プログラムの例　filestreamkaku.nako

```
入力パス名とは文字列　＝　「c:\*.*」でファイル選択
もし入力パス名が「」でないならば
  出力パス名とは文字列　＝　「c:\*.*」で保存ファイル選択
  もし出力パス名が「」でないならば
    ハンドルとは整数　＝　出力パス名を「書」でファイルストリーム開く
    入力パス名を毎行読んで反復
      ハンドルで回数＆「: 」＆それ＆改行をファイルストリーム書く
    ハンドルをファイルストリーム閉じる
```

12.4 ファイルの列挙

12.4.1 ファイルの列挙の基礎

この節では、フォルダの中にあるファイルを列挙する命令、つまり、フォルダの中にあるファイルの一覧を取得する命令を紹介したいと思います。

フォルダの中にあるファイルの一覧は、「ファイル列挙」という命令を使うことによって、取得することができます。

「ファイル列挙」は、引数としてフォルダのパス名を受け取って（助詞は「の」）、そのフォルダの中にあるファイルの名前から構成される配列を返します。

次のプログラムは、指定されたフォルダの中にあるファイルの一覧をダイアログボックスで表示します。

プログラムの例　filerekkyo.nako

```
パス名とは文字列　＝　「c:\」でフォルダ選択
パス名が「」でないならば
　ファイル一覧とは配列　＝　パス名のファイル列挙
　ファイル一覧を改行で配列結合をメモ記入
```

12.4.2　ワイルドカードによる絞り込み

「ファイル列挙」に引数として渡すフォルダのパス名の末尾に、ファイル名のパターンを示すワイルドカードを置くことによって、そのワイルドカードによってファイルの一覧を絞り込むことができます。たとえば、*.txt というワイルドカードを末尾に置いたパス名を引数として「ファイル列挙」に渡すことによって、ファイルの一覧を、.txt という拡張子を持つものに絞り込むことができます。

「ファイル列挙」に渡す引数の末尾には、セミコロン (;) で区切ることによって、ワイルドカードを何個でも置くことができます。たとえば、

```
c:\windows\*.exe;*.dll
```

というパス名を引数として「ファイル列挙」に渡すことによって、c:\windows というフォルダの中にあるファイルのうちで、.exe または .dll という拡張子を持つものだけの一覧を取得することができます。

次のプログラムは、指定されたフォルダの中にあるファイルのうちで、.exe または .dll という拡張子を持つものの一覧をダイアログボックスで表示します。

プログラムの例　wildcard.nako

```
パス名とは文字列　＝　「c:\」でフォルダ選択
パス名が「」でないならば
　パス名改とは文字列　＝　パス名＆「*.exe;*.dll」
　ファイル一覧とは配列　＝　パス名改のファイル列挙
　ファイル一覧を改行で配列結合をメモ記入
```

12.4.3　フォルダの列挙

「ファイル列挙」を使うことによって、フォルダの中にあるファイルの一覧を取得することができるわけですが、ファイルではなくてフォルダの一覧を取得することは、この命令にはできません。

フォルダの中にあるフォルダの一覧は、「フォルダ列挙」という命令を使うことによって、取得することができます。

「フォルダ列挙」は、引数としてフォルダのパス名を受け取って（助詞は「の」）、そのフォルダの中にあるフォルダの名前から構成される配列を返します。

次のプログラムは、指定されたフォルダの中にあるフォルダの一覧をダイアログボックスで表示します。

プログラムの例　folderrekkyo.nako

```
パス名とは文字列　＝　「c:\」でフォルダ選択
パス名が「」でないならば
　フォルダ一覧とは配列　＝　パス名のフォルダ列挙
　フォルダ一覧を改行で配列結合をメモ記入
```

12.4.4　再帰的なファイルまたはフォルダの列挙

「ファイル列挙」と「フォルダ列挙」は、指定されたフォルダの中にあるファイルまたはフォルダの一覧を返すわけですが、この場合の「中にある」というのは、厳密に言えば、指定されたフォルダよりも一段階だけ下の階層にある、ということです。したがって、指定されたフォルダの二段階よりも下の階層にあるものは、一覧には含まれません。

一段階だけ下の階層にあるものの一覧ではなくて、再帰的に、その下やさらにその下にある

フォルダも調べることによってできるファイルまたはフォルダの一覧は、「全ファイル列挙」または「全フォルダ列挙」という命令を使うことによって取得することができます。

「全ファイル列挙」は、引数としてフォルダのパス名を受け取って（助詞は「の」）、そのフォルダよりも下のすべての階層にあるファイルを再帰的に調べて、それらのファイルの絶対パス名から構成される配列を返します。

同じように、「全フォルダ列挙」は、引数としてフォルダのパス名を受け取って（助詞は「の」）、そのフォルダよりも下のすべての階層にあるフォルダを再帰的に調べて、それらのフォルダの絶対パス名（末尾は常にバックスラッシュ）から構成される配列を返します。

次のプログラムは、指定されたフォルダよりも下のすべての階層にあるファイルの一覧をダイアログボックスで表示します。

プログラムの例　zenfilerekkyo.nako

```
パス名とは文字列　＝　「c:\」でフォルダ選択
パス名が「」でないならば
  ファイル一覧とは配列　＝　パス名の全ファイル列挙
  ファイル一覧を改行で配列結合をメモ記入
```

次のプログラムは、指定されたフォルダよりも下のすべての階層にあるファイルの一覧をダイアログボックスで表示します。

プログラムの例　zenfolderrekkyo.nako

```
パス名とは文字列　＝　「c:\」でフォルダ選択
パス名が「」でないならば
  フォルダ一覧とは配列　＝　パス名の全フォルダ列挙
  フォルダ一覧を改行で配列結合をメモ記入
```

次のプログラムは、指定されたフォルダよりも下にあるすべての階層のファイルのうちで、拡張子が .nako のものについて、それらの内容を連結した結果をダイアログボックスで表示します。

プログラムの例　zenfileketsugou.nako

```
パス名とは文字列　＝　「c:\」でフォルダ選択
パス名が「」でないならば
  パス名改とは文字列　＝　パス名＆「*.nako」
  ファイル一覧とは配列　＝　パス名改の全ファイル列挙
  甲とは文字列　＝　「」
  乙とは文字列
  丙とは整数
  丙で０から（ファイル一覧の配列要素数から１を引く）まで繰り返す
    乙にファイル一覧￥丙から読む
    甲は甲＆乙
  甲をメモ記入
```

第 13 章　ネットワーク

13.1　ネットワークの基礎

13.1.1　プロトコル

なでしこは、ネットワークによる通信のためのさまざまな機能を持っています。この章では、なでしこが持っているネットワークの機能について説明したいと思います。

ネットワークを介してコンピュータとコンピュータ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」と呼ばれます。インターネットで使われるプロトコルは、IETF(Internet Engineering Task Force) という組織によって策定されています[1]。

IETF は、標準として策定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、RFC 1149 とか RFC 2324 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]。

[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。

13.1.2　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は、「ホスト」と呼ばれます。ネットワークを介して通信をするためには、通信の相手とな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」と呼ばれる番号を持っています（RFC 791）。

IP アドレスは、32 個のビットから構成される列です。それを文字列で記述するときは、通常、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
```

という文字列で記述されます。

13.1.3　ドメイン名

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、ホストの識別には、英字や数字などを使って作られた「ドメイン名」と呼ばれる名前を使うこともできるようになっています。

ドメイン名は、「ラベル」と呼ばれる名前をドットで区切って並べた、

ラベル . ラベル . ・・・ . ラベル

という構造を持つ名前です。ラベルを作るために使うことのできる文字は、英字と数字とハイフンです。

ドメイン名を構成しているそれぞれのラベルは、右から左へ順番に、トップレベルドメイン、第 2 レベルドメイン、第 3 レベルドメイン、……と呼ばれます。たとえば、

```
www.example.com
```

というドメイン名の場合、トップレベルドメインは `com` で、第 2 レベルドメインは `example` で、第 3 レベルドメインは `www` です。

13.1.4　DNS

インターネットには、IP アドレスとドメイン名とを対応させる、DNS と呼ばれるシステムがあります。ですから、IP アドレスを知らなくても、ドメイン名さえ分かれば、DNS を使うことによって、それを IP アドレスに変換することができます。

なでしこでは、「IP アドレス取得」という命令を使うことによって、ドメイン名を IP アドレスに変換することができます。「IP アドレス取得」は、引数としてドメイン名を受け取って（助詞は「の」）、そのドメイン名に対応する IP アドレスをあらわす文字列を戻り値として返します。

次のプログラムは、ダイアログボックスでドメイン名を読み込んで、そのドメイン名を IP アドレスに変換した結果をダイアログボックスで表示します。

プログラムの例　`ipaddressshutoku.nako`

```
ドメイン名とは文字列　＝　「ドメイン名」と尋ねる
ドメイン名の IP アドレス取得を言う
```

13.2　HTTP

13.2.1　HTTP の基礎

「ウェブ」または「WWW」(world wide web) と呼ばれるサービスは、「ウェブサーバー」と呼ばれるプログラムと、「ウェブブラウザー」または単に「ブラウザー」と呼ばれるプログラムとのあいだの通信によって成立します。

ウェブサーバーとウェブブラウザーとのあいだの通信には、HTTP(Hypertext Transfer Protocol) と呼ばれるプロトコル（RFC 2616）が使われます。

13.2.2 ファイルへのダウンロード

「HTTP ダウンロード」という命令を使うことによって、ウェブサーバーからデータをダウンロードして、そのデータをファイルに保存する、ということができます。

「HTTP ダウンロード」は、引数として URL（助詞は「を」）とパス名（助詞は「へ」）を受け取って、URL で指定されたデータをダウンロードして、パス名で指定されたファイルに保存します。

次のプログラムは、URL で指定されたデータをダウンロードして、指定されたファイルにそのデータを保存します。

プログラムの例 `httpdownload.nako`

```
URL とは文字列 = 「URL」と尋ねる
パス名とは文字列 = 「c:\*.*」で保存ファイル選択
URL をパス名へ HTTP ダウンロード
```

13.2.3 ウェブサーバーからのデータの取得

ウェブサーバーからデータをダウンロードする命令としては、「HTTP ダウンロード」のほかに、「HTTP データ取得」という命令もあります。

「HTTP データ取得」は、引数として URL を受け取って（助詞は「を」）、指定されたデータをダウンロードして、そのデータを戻り値として返します。

次のプログラムは、URL で指定されたデータをダウンロードして、そのデータをダイアログボックスで表示します。

プログラムの例 `httpdatashutoku.nako`

```
URL とは文字列 = 「URL」と尋ねる
URL を HTTP データ取得をメモ記入
```

13.2.4 ステータス行とレスポンスヘッダの取得

ウェブブラウザーがウェブサーバーに対してデータの要求を送信すると、ウェブサーバーは、「レスポンス」と呼ばれるデータをウェブブラウザーに送ります。レスポンスは、「ステータス行」と呼ばれる行と、「レスポンスヘッダ」と呼ばれるいくつかの行と、そしてデータの本体から構成されます。

「HTTP ダウンロード」がファイルに保存するデータや、「HTTP データ取得」が戻り値として返すデータは、レスポンスの全体ではなくて、データの本体だけです。

ステータス行とレスポンスヘッダは、「HTTP ヘッダ取得」という命令を使うことによってウェブサーバーから取得することができます。

「HTTP ヘッダ取得」は、引数として URL を受け取って（助詞は「を」）、指定されたデータをウェブサーバーに要求して、ウェブサーバーから送られてきたレスポンスに含まれているステータス行とレスポンスヘッダーを戻り値として返します。

次のプログラムは、URL で指定されたデータをウェブサーバーに要求して、ウェブサーバーから送られてきたレスポンスに含まれているステータス行とレスポンスヘッダーをダイアログボックスで表示します。

プログラムの例 `httpheadershutoku.nako`

```
URL とは文字列 = 「URL」と尋ねる
URL を HTTP ヘッダ取得を言う
```

13.3 SMTP

13.3.1 SMTP の基礎

メールは、送信者から受信者にまで届く過程で、「メール転送エージェント」(MTA) と呼ばれるプログラムのあいだで何度か転送が繰り返されます。メール転送エージェントが別のメール転送エージェントへメールを転送するときには、SMTP(Simple Mail Transfer Protocol) と呼ばれるプロトコル（RFC 5321）が使われます。

ユーザーがメールの送受信に使うプログラムは、「メールユーザーエージェント」(MUA) と呼ばれます。SMTP は、メール転送エージェントが別のメール転送エージェントへメールを転送す

るときだけではなくて、メールユーザーエージェントがメール転送エージェントにメールを送信するときにも使われます。

13.3.2　メールを送信するために必要となるデータ

なでしこのプログラムが SMTP でメールを送信する場合には、送信に先立って、なでしこによって宣言されているいくつかの変数に対して、メールを送信するために必要となるデータを代入しておくことが必要になります。

メールの送信に必要なデータを代入する変数としては、次のようなものがあります。

メールホスト	メールの送信先となるホストのホスト名。
メール差出人	メールの差出人のメールアドレス。
メール宛先	メールの宛先のメールアドレス。
メール件名	メールの件名。
メール本文	メールの本文。
メール CC	メールをカーボンコピーで送る宛先のメールアドレス。二人以上の人に送る場合は、メールアドレスをコンマで区切って並べる。
メール BCC	メールをブラインドカーボンコピーで送る宛先のメールアドレス。二人以上の人に送る場合は、メールアドレスをコンマで区切って並べる。
メール添付ファイル	メールに添付するファイルのパス名。2 個以上のファイルを添付する場合は、パス名を改行で区切って並べる。

これらの変数のうち、「メール CC」と「メール BCC」と「メール添付ファイル」については、必要がなければデータを代入しなくてもかまいません。

13.3.3　メールを送信する命令

「メール送信」という命令は、先ほど紹介した変数に代入されているデータにもとづいて、SMTP を使ってメールをメール転送エージェントへ送信します。

次のプログラムは、メールホストのホスト名、宛先のメールアドレス、差出人のメールアドレス、件名、本文を読み込んで、メールを送信します。ただし、ユーザー認証をしないとメールの送信ができないようにプロバイダーがメールホストを設定している場合、このプログラムでメールを送信することはできません。

プログラムの例　mailsoushin.nako

```
甲とはラベル
甲について
  位置は「40,20」
  テキストは「ホスト」
ホストとはエディタ
ホストについて
  位置は「90,20」
  幅は 400
  注目
乙とはラベル
乙について
  位置は「40,50」
  テキストは「宛先」
宛先とはエディタ
宛先について
  位置は「90,50」
  幅は 400
丙とはラベル
丙について
  位置は「40,80」
  テキストは「差出人」
差出人とはエディタ
差出人について
  位置は「90,80」
  幅は 400
丁とはラベル
丁について
```

```
  位置は「40,110」
  テキストは「件名」
件名とはエディタ
件名について
  位置は「90,110」
  幅は 400
戊とはラベル
戊について
  位置は「40,140」
  テキストは「本文」
本文とはメモ
本文について
  位置は「90,140」
  幅は 400
  高さは 170
送信とはボタン
送信について
  位置は「90,320」
  幅は 200
  テキストは「メールを送信する」
  クリックした時は
    メールホストはホストのテキスト
    メール差出人は差出人のテキスト
    メール宛先は宛先のテキスト
    メール件名は件名のテキスト
    メール本文は本文のテキスト
    メール送信
```

索引