ドリトル言語の基礎知識

ここでは、最初に「プログラミングとはどういうことか」ということを簡単に説明した後、ドリトル言語によるプログラムの基本的な概念と原理を説明する。

プログラミングとは

我々が普段コンピュータを使っているとき、その上ではさまざまなプログラムが動いていて、我々の相手をしてくれる（Webブラウザであればネットワーク上からさまざまな情報（HTMLファイルや画像ファイル）を取り寄せて画面に表示してくれるし、ワープロソフトであれば打ち込んだ文字をさまざまに整形して、画面上にそれらしく配置し、またそれをプリンタから印刷させてくれる。

では、これらの「プログラム」の中身は具体的にどのようなものなのだろうか? また、なぜこれらの「プログラム」を取り替えるだけで、コンピュータはさまざまな作業をこなしてくれるのだろうか?

これらの答えは実は1つのこと、つまり「コンピュータとは命令を順番に実行していく装置（機械）であり、それぞれの作業をこなすように命令を並べたものがプログラムである」ということに落ち着く。これを先の2つの問いの答えの形にまとめると、次のようになる。

- プログラムは、コンピュータに実行させる命令の並びで、その並びを実行させることでコンピュータにそのプログラムとしての動作を行わせる。
- コンピュータは単なる命令を実行する装置だから、その命令を取り替えることによって、どのような作業でもこなすようにさせられる。

そして、プログラミングとは「プログラムを作ること」であり、上の説明の延長でいえば、コンピュータに「自分がさせたいこと」を実行させるように、命令を並べて行く作業だ、ということになる。

では、具体的にどのような命令を並べて行けばいいのだろうか? コンピュータはすべての情報を「0」と「1」の組合せから成るビット列（ディジタル情報）として表すので、一番根本部分ではCPU（中央処理装置のレベルでは）が命令を表すビットの列（機械語）を並べたものを実行していく。

しかし、「0」と「1」の組み合せでプログラムを作るのは大変繁雑で間違いやすいので、今日ではまず行われない。その代わり、人間に分かりやすい書き方で命令を書き表す。この書き表し方のことをプログラミング言語といい、いくつもの種類がある。

プログラミング言語で書き表したものを実際に実行するためには、また別のソフトウェア（プログラミング言語処理系）を使う。プログラミング言語処理系には、プログラミング言語で書き表したものを機械語に変換する方式（コンパイラ）と、プログラミング言語で書き表したものを読み取りながら直接その動作を実行する方式（インタプリタ）とがある。

ここまでは一般的な話だったが、次節以降では命令の内容や書き表し方としてドリトル言語の場合を学んで行くことにする。現在のドリトルのプログラミング言語処理系はインタプリタ方式を採用している。

オブジェクト、プロパティ、変数

一般にプログラムでは、さまざまな情報（データ）を取り扱う。ドリトルでは、これらを総称して**オブジェクト**（もの）と呼んでいる。我々が日常生活で接する「もの」はさまざまな大きさ、形、色、機能を持っているが、ドリトルのオブジェクトも同様である。ドリトルはコンピュータ上で動くプログラミ

ング言語なので、これらの大きさ、色、機能なども、それらをプログラムで扱うとしたら、すべてコンピュータの上の情報として表すことになる…つまり、大きさ、色、機能などの情報もまたもとのオブジェクトに付随するオブジェクトだということになる。このような、オブジェクトに付随する情報をそのオブジェクトの**プロパティ**と呼ぶ。

このように、すべての情報をオブジェクトとして統一的に扱い、さまざまな機能もオブジェクトに付随しているという形で扱うプログラミング言語を**オブジェクト指向言語**という。今日のソフトウェア開発ではオブジェクト指向言語が多く使われる□C++□Javaなどはオブジェクト指向言語の例である。ドリトルもオブジェクト指向言語である。

プログラム中でさまざまなオブジェクトを扱う上で、それらに**名前**が付けられていないと不便である。ドリトルでは、オブジェクトに直接名前を付けるのではなく、オブジェクトを入れる「いれもの」に名前がついている。この「いれもの」のことを**変数**という。

ドリトルでは、変数にオブジェクトを格納する動作（**代入**）を「**=**」で表す。たとえば、次のコードでは「**ボタン1**」という名前の変数に、「=」の右側で指定したオブジェクトを格納することになる（指定のしかたは次の節で説明する）。

```
ボタン1 = ...。
```

変数は「いれもの」なので、プログラムの中でさまざまに内容（オブジェクト）を入れ換えることもできる。ある変数の内容を入れ換えないで、ずっと1つのオブジェクトを入れておく場合には、その変数の名前のことを、その「オブジェクトの名前」だと思っていても差し支えない。

ドリトルでは、さまざまなプログラムで使う標準的なオブジェクトが、決まった一連の変数に最初から格納されていることになっている。これらの変数は普通、書き換えないので、「さまざまな標準オブジェクトがあり、それらは固有の名前を持っている」と考えていてよい。

オブジェクトに付随するプロパティも、それぞれ名前を持っている。変数に格納されているオブジェクトのプロパティを指定する1つの方法は、次のように、変数名の後に「**:**」で区切ってプロパティ名を指定することである。

```
ボタン1：動作 = ...。
```

この例でも分かるように、ドリトルではオブジェクトのプロパティを変数と同じように扱うことができる。

メッセージ送信

前節で、オブジェクトに付随するもののなかに「機能」があると述べた。これらの機能も、オブジェクトにプロパティとして付属しているので、そのプロパティ名を用いて「この機能」ということを指定できる。オブジェクト指向言語では通常、機能のことを**メソッド**と呼ぶので、本書でも以下こちらの呼び方を用いる。メソッドとは要するにオブジェクトが持つ（オブジェクトに付随する）機能のことだと覚えておいて頂ければよい。オブジェクトのメソッドを使うということは、通常はそのメソッドを「呼び出す（動かす、働かせる）」ことである。

たとえば、**ボタン**という名前の標準オブジェクトがあるが、それに付属している「作る」というメソッドを呼び出すと、新しいラベル（表示欄）オブジェクトが作られて返される。そのようにして新しいラベルを作り、それを変数「ボタン1」に格納するには、次のようにする。

```
ボタン1 = ボタン！作る。
```

このような、「オブジェクト ！ メソッド名」という書き方を、オブジェクトに呼びかけることになぞらえて**メッセージ送信**と呼ぶ。また、最後の「。」は、ここまででひとまとまりの動作（文）が終ることを表す。オブジェクトを省略するか**自分**と書いた場合には、そのメソッドを実行しているオブジェクトにメッセージが送られる。

メッセージ送信によりオブジェクトのメソッドを呼び出すとき、必要に応じて追加の情報を渡すことができる。これをメソッド呼び出しの**パラメータ**と呼ぶ。ドリトルでは、パラメータはメソッド名の**前に**書くことになっている。

たとえば、ボタンのメソッド「作る」には、パラメータとしてボタンに表示するラベルの文字列を渡すことができる。先のプログラムをそのように直してみる。

```
ボタン1 = ボタン！"テスト" 作る。
```

このプログラム実行すると、先程と同様にボタンが現れるが、そのラベルが「テスト」になっている。

すべてのメソッドは、実行した後、何らかの値（オブジェクト）を返す。1つのメッセージ送信（メソッド呼び出し）の後に、続けてまた別のメソッド名を書くことで、前のメソッドが返したオブジェクトに対するメソッド呼び出しを行うことができる。これをメッセージの**カスケード**（直列接続）と呼ぶ。

たとえば、ボタンを作った後、その位置を指定するには、たとえば次のようにして、ボタンのメソッド「位置」を呼び出す。

```
ボタン1 = ボタン！"テスト" 作る 100 50 位置。
```

上の例のように、メソッドのパラメータは複数個指定してもよい。では、並んでいるものがメソッドの名前なのかパラメータなのかはどうやって分かるのだろうか? 実は、「！」の右側ではすべての名前はメソッド名として扱われる。「“ “」で囲まれた文字列や、「100」などの数値はメソッド名ではないのでパラメータになる。変数名を指定したいときは、変数名を「（）」で囲んで□□□□□などのように指定する（後の節で詳しく説明する）。

メッセージのカスケードがある場合、変数に格納される値（メッセージ送信式全体の値）は、一番最後のメソッドが返した値になる。上のプログラムの場合、メソッド「位置」はボタンの位置を変更した後、そのボタンオブジェクト自体を返すので、変数「ボタン1」に格納される値は先のプログラムと同じである。

それぞれのメッセージが何を返すかは、メソッドの定義内容によって異なる。しかしそれでは分かりにくいので、ドリトルの標準定義のメソッドでは、元のオブジェクトをそのまま返すものが多い。ただし、新しいオブジェクトを作り出すことが目的の「作る」「～を作る」という名前のメソッドと、「？」で終る名前を持つオブジェクトのさまざまな性質を調べることが目的のメソッドは、この原則の例外になっている。

中置記法

ドリトルでさまざまな処理を指定する書き方の1つは上で説明したメッセージ送信であるが、もう1つの書き方として**中置記法**がある。中置記法では□□□1□などのように、計算対象の間に演算を書くことで、加減乗除などの演算を読みやすく書くことができる。なお、この例をメッセージ送信記法で書くと□□□1足す」になる。

ドリトルでは1つの式は全体としてメッセージ送信記法であるか、全体として中置記法であるかのどちらかであるが、「（…）」で囲まれた中は外側とは別にメッセージ送信記法か中置記法かを選ぶことができる。メッセージ送信記法か中置記法かは、「！」の有無で判断できる。

中置記法とメッセージ送信記法のもう1つの違いは、中置記法の中では名前は変数を表す、ということである。これに対し、メッセージ送信記法の中では直接変数を書けるのは「！」の左側だけであり、「！」の右側では名前はメソッド名として扱われる。したがって、パラメータなどで変数を指定したい場合は「（…）」で囲む必要がある（囲んだ中は中置記法にできるため）。たとえば、中置記法の「□□□□をメッセージ送信記法で書くと□□□□□□足す」になるが、この□□□□□は丸かっこの中に（「！」がないため）中置記法が書かれているものとして扱われている。

中置記法の中で使える演算子としては、+（足す）、-（引く）、□（掛ける）、/（割る）、%（余り）がある（かっこ内はメソッドとして使うための名称）。このほか、比較演算子==□eq□□!=□ne□□

>**□gt□□**>=**□ge□□**<**□lt□□**<=**□le□もある。

また、中置記法の中では関数記法が使える。たとえば□□sqrt□2□□は「2□sqrt□の略であり、そのほか任意のパラメータなしのメソッド「オブジェクト！メソッド」を「メソッド（オブジェクト）」の形で書くことができる。

中置記法を使う場合には全体を括弧「（…）」で囲む必要があるが、代入文の右辺では括弧を省略して□□x = x + 1□のように書くことも可能である。

ブロックとメソッド

角かっこ（「…」または […]）は、ドリトルでは**ブロック**を表すものとして扱われる。ブロックとはプログラムを部品化する仕組みであり、一連の動作を「後で実行したり、繰り返し実行するためにとっておく」ものと考えることができる。ブロックのメソッド**実行**によって取っておかれた動作を実行させられる（以下で出てくるが、これ以外にもブロックの動作を実行させる方法は多数ある）。次の例はブロックを2回実行するので、ラベルに30が表示される。

```
  □ □ 10□
動作1□□□□□□10□□
動作1！実行。
動作1！実行。
 ラベル□□□□作る。
```

実際には、ブロックはもっと「とっておく」価値のある場面で使われる。たとえば、次のプログラムを考えてみる。

```
ボタン1 ＝ ボタン！"テスト" 作る 10 50 位置。
ボタン1□□ □ 10□
ボタン1：動作 □ □□ □ □ □ 10□ボタン1□□□□50 位置」。
```

このプログラムでは、前の例題と同様にボタンを作った後、ボタンのプロパティ□に10を格納し、またボタン1の「動作」というプロパティにブロックを格納している。

オブジェクトのプロパティとしてブロックを格納すると、そのプロパティはオブジェクトのメソッドになる。そして、ボタンは押されるとそれ自身の「動作」というメソッドを呼び出すように作られているので、ボタンを押すたびにこのブロックが実行される。

ブロックがメソッドになっているときは、その中では□x等はそのメソッドを持っているオブジェクトのプロパティxに対応する（後の節で詳しく説明する）。ここでは、まずxに格納されている値に10を足し

た値を計算し、その結果を再びxに格納している。次に、ボタンの位置をX座標がxの値、Y座標が50になるように変更している。これにより、ボタンは押されるごとに位置が右に10ずつ移動することになる。

ブロックの先頭に「｜…｜」で囲んでパラメータを書くことで、メソッドを呼び出すときに渡されたパラメータを受け取ることができる。ブロックのパラメータは、渡された値を初期値として持つ**ローカル変数**として扱われる。ブロックのパラメータは、「実行」でブロックを動作させるときにも渡すことができる。（ローカル変数は後の節で詳しく説明する）

```
かめた＝タートル！作る。
かめた：曲がって動く ＝ 「｜角度 距離｜！（角度）右回り（距離）歩く」。
□ □ かめた！30 100 曲がって動く 横の位置？。
ラベル□□□□作る。
```

「！」の前に何もないときは、そのメソッドを持っているオブジェクトが指定されているものとして扱われる。また、特別な名前「自分」もメソッドを持っているオブジェクトを指定するのに使うことができる。従って上の例は「自分！…」のように書いてもよい。

ブロックは1つの式であり、最後に評価した式の値をブロック全体の値として返す。メソッドから値を返したいときはこれを利用する。上の例では、最後に評価されるのはメソッド「歩く」なので、それが返した値、つまりかめた自身が「曲がって動く」の返す値となる。そこで引き続きかめたに対するメソッド「横の位置？」を呼び出してそのX座標を調べているわけである。

ブロックと制御構造

ドリトルのブロックは、メソッドを作ることのほかに、さまざまな制御構造を実現するのにも使われる。それには、ブロックオブジェクト自身が持つメソッドを使うのが基本である。以下で説明する。

指定回数の繰り返し

ブロックのメソッド**繰り返す**を使うことで、そのブロックに書かれた動作を指定回数、繰り返し実行することができる。繰り返し回数（任意の数値）はメソッドのパラメータとして指定する。

```
□...□□n□繰り返す。
```

次のプログラムを実行すると、初期値が0の変数□a□に1を加える処理を10回行い、結果として画面に10が表示される。

```
a=0□
□a=a+1□□10□繰り返す。
ラベル□□a□作る。
```

現在が何回目かの実行であるかを、ブロックのパラメータとして受け取ることができる。

```
□|i| ...□□n□繰り返す。
```

次のプログラムを実行すると、初期値が0の変数□a□に、何回目の繰り返しかを示すパラメータ□i□を加える処理を10回行い、結果として画面に55が表示される□□i□の値は、繰り返しを実行するたびに 1, 2, 3, …, 10 と変化する。その結果□□a=a+i□を実行するたびに□a□に 1, 2, 3, …, 10が順に足されることに

なり、結果として「0 + 1 + 2 + … + 10」の合計が表示される。

```
a=0□
□|i| a=a+i□□10□繰り返す。
ラベル□□a□作る。
```

条件が成り立つ間の繰り返し

最初に回数を指定するのではなく、条件を指定して、その条件が成り立っている間繰り返す場合には、ブロックのメソッドの間と実行を次のように組み合わせて使う。

```
「…」！の間「…」実行。
```

たとえば、100以上の値を持つ最初のフィボナッチ数を表示させるプログラムは次のようになる。

```
x1=1□x2=1□
□x1 □ 100□□の間「z=x2+x1□x1=x2□x2=z□実行。
ラベル□□x1□作る。
```

ここで「の間」は何をするメソッドかというと、1番目のブロックを中に保持して、繰り返し調べる準備をしたオブジェクトを返す。このオブジェクトに対して2番目のブロックをパラメータとしてメソッド「実行」を呼び出すと、「1番目のブロックの実行」→「結果が真であれば続行」→「2 番目のブロックの実行」→「1番目のブロックの実行」→「結果が真であれば続行」→… のように繰り返しが続いて行く。

条件分岐

条件分岐（枝分かれ）は、ある条件の成否に応じてプログラムの一部を実行する。条件分岐もブロックのメソッドならとそうでなければと実行を組み合わせて記述する。

```
「…」！なら「…」実行。
「…」！なら「…」そうでなければ「…」実行。
```

いずれも、1番目のブロックを実行した結果が真であれば2番目のブロックが実行される。さらに下の形では、2番目のブロックが実行されなかった場合は3番目のブロックが実行される。メソッド「なら」や「そうでなければ」はこれらの制御を適宜行うためのオブジェクトを返す。

次のプログラムを実行すると、最初に1から10までの乱数を発生し、変数「数」に入れる。続いて、「数」が5より大きいかを判定し、真のときは「なら」に続くブロックを実行し、偽のときは「そうでなければ」に続くブロックを実行する。結果として、実行するたびに、画面に「大きい」と「小さい」がランダムに表示される。

```
数＝乱数（10）。
「数 ＞ 5」！なら「ラベル！"大きい" 作る」
  そうでなければ「ラベル！"小さい" 作る」実行。
```

上の例では、ブロックの中でラベルを作ることで結果を画面に表示した。ブロックを実行すると、最後に実行された値がブロックの実行結果として返される。次のプログラムでは、「なら」に続くブロックと

「そうでなければ」に続くブロックから文字列を返し、その値を「結果」という変数に入れている。そして、その値をラベルで表示している。

```
数=乱数（10）。
結果=「数 > 5」！なら「"大きい"」そうでなければ「"小さい"」実行。
ラベル！（結果）作る。
```

他の言語で「if-elseの連鎖」と呼ばれる、複数の条件を順に調べて行く形の枝分かれは「そうでなければ」の後に次に調べる条件を書くことで記述できる。つまり、次の形になる。

```
「条件1」！なら「...」そうでなければ「条件2」なら「...」
  そうでなければ「条件3」なら「...」そうでなければ「...」実行。
```

タイマーとスレッド

動作の実行に遅延を設けたり、一定時間間隔で繰り返し実行させたい場合には、**タイマーオブジェクト**を利用する。

タイマーによる実行

タイマーは、あらかじめ指定された**間隔**で、指定された**回数**または**時間**だけ、**ブロック**を繰り返して実行する。標準では、間隔が0.1秒、回数が100回に設定されている。

ブロックを指定して**タイマー**の**メソッド実行**を呼び出すと、**タイマー**は**パラメータ**として受け取った**ブロック**を設定された時間間隔で繰り返し実行する。最初の実行は、指定された間隔だけ待った後に行われる。

```
タイマー！「...」実行。
```

次の**プログラム**を実行すると、「はろー」という文字が、画面で少しずつ右下に動いていく。**移動する**は、画面の**オブジェクト**を動かす命令である。「3 -2 移動する」は、**オブジェクト**を現在の位置から「右に3」、「下に2」だけ移動するという意味である。これを一定間隔で繰り返して実行することにより、文字は少しずつ画面を移動していくことになる。

```
表示＝ラベル！"はろー"　作る。
時計＝タイマー！作る。
時計！「表示！3　-2　移動する」実行。
```

タイマーの動作設定

繰り返す間隔は、「時計！0.2　間隔。」のように「間隔」**メソッド**で指定する。単位は秒である。

繰り返す回数は、「時計！10　回数。」のように「回数」**メソッド**で指定する。

次の**プログラム**を実行すると、画面の文字が右下に移動する。間隔を0.5秒に設定したので、標準である0.1秒の間隔と比べると、ギクシャクとした動きになる。いちどに動く距離は、直前の例と比べて10倍に大きくした。回数は10回に設定したので、5秒間動くと終了する。

```
表示＝ラベル！"はろー"　作る。
時計＝タイマー！作る　0.5　間隔　10　回数。
時計！「表示！30　-20　移動する」実行。
```

アニメーションやゲームなどのプログラムを作る場合、タイマーを実行する間隔は、標準の0.1秒では少しぎこちなく見えるかもしれないが、放送で使われる映像が0.02秒から0.04秒程度の間隔であることから、ほとんどの場合は標準の0.1秒から0.02秒程度が適切である。これより短く設定することも可能だが、システムへの負荷を考慮して、最小の間隔である0.001秒（1ミリ秒）より短く設定できないようになっている。

タイマーの動作は、間隔と回数で指定するほかに、間隔と時間で指定することもできる。繰り返す時間は、「時計！10　時間。」のように「時間」メソッドで指定する。単位は秒である。

通常、「0.1秒間隔で100回実行」と「0.1秒間隔で10秒間実行」の動作は同じだが、繰り返し間隔より時間のかかる動作を実行する場合には動作が異なる可能性がある。たとえば、いちどに0.2秒程度かかる動作を0.1秒間隔で実行した場合は次のような処理が行われる。

- 0.2秒かかる処理を0.1秒間隔で100回実行した場合は、約20秒かけて100回の実行が行われる。
- 0.2秒かかる処理を0.1秒間隔で10秒間実行した場合は、10秒かけて約50回の実行が行われる。

つまり、「必ず指定した回数を実行させたい」ときは回数を指定し、「必ず時間内に実行を終らせたい」ときは時間を指定すればよい。

繰り返し回数の利用

現在が何回目の実行かは、繰り返しと同様に、ブロックのパラメータとして受け取ることができる。次のプログラムを実行すると、繰り返すたびに移動する距離が長くなり、結果として進む速度が速くなったように見える。

```
表示＝ラベル！"はろー"　作る。
時計＝タイマー！作る。
時計□□|i|　表示□□i□0□移動する」実行。
```

タイマーの終了を待って実行する

ドリトルのプログラムは通常、上から順に実行され、ひとつの文の実行が終るのを待って次の文の実行が行われる。

これに対し、タイマーの実行は、終わるまでに標準の設定では10秒間かかる。この間に他のプログラムの実行を行えないと、全体としてプログラムの動作が止まってしまうことになる。

そこで、タイマーの実行は、ドリトルのプログラムと並行して実行されるようになっている。タイマーの実行が終わるのを待ってから、さらにタイマーで実行したいときは「次に実行」を指定する。タイマーの実行が終わるのを待ってから、次に1回だけ実行したいときは「最後に実行」を指定する。

次のプログラムを実行すると、時計の実行が開始されると同時に最後の行のプログラムが実行され、画面に「こんにちは」というメッセージが表示される。

```
表示＝ラベル！"はろー"　作る。
時計＝タイマー！作る。
時計！「表示！3　-2　移動する」実行。
 ラベル！"こんにちは"　作る。
```

次のプログラムを実行すると、時計の実行が終了するのを待ってから、画面に「こんにちは」というメッ

セージが表示される。

```
表示＝ラベル！"はろー"　作る。
時計＝タイマー！作る。
時計！「表示！3　-2　移動する」実行。
時計！「ラベル！"こんにちは"　作る」最後に実行。
```

次のプログラムを実行すると、時計の実行が終了するのを待ってから次の時計の実行が行われ、それらが終了するのを待ってから最後のプログラムが実行される。

```
表示＝ラベル！"はろー"　作る。
時計＝タイマー！作る。
時計！「表示！3　-2　移動する」実行。
時計！「表示！3　2　移動する」次に実行。
時計！「ラベル！"こんにちは"　作る」最後に実行。
```

配列と複数オブジェクトの利用

配列オブジェクトを使うと、その中に複数のオブジェクトを入れておき、取り出して使うことができる。入れるオブジェクトの数や種類は、あらかじめ決めておく必要はない。

配列の生成と参照

配列は、あらかじめ入れておきたいオブジェクトをパラメータで指定して「作る」を実行する。配列の要素は、1から始まる番号をパラメータに指定して、**読む**で読み出すことができる。

次のプログラムでは、「並び」という名前の配列を生成し、3個の数値を入れている。そして、3番目の要素を画面に表示している。

```
並び＝配列！123　456　789　作る。
ラベル！（並び！3　読む）作る。
```

配列に入っている要素の数は、**要素数？**で調べることができる。次のプログラムでは、配列に3個の数値を入れた後で、要素数を画面に表示している。画面には3が表示される。

```
並び＝配列！123　456　789　作る。
ラベル！（並び！要素数？）作る。
```

要素の追加・更新・削除

書くを使うと、配列の末尾に新しい要素を追加することができる（要素数は1多くなる）。また、**上書き**を使うと、配列の指定位置の要素を書き換えることができる（要素数は変わらない）。**消す**を使うとオブジェクトを指定して、**位置で消す**を使うと要素の位置を指定して、要素を削除することができる。

次のプログラムでは、3つの要素が格納された配列の末尾に要素を追加し、3番目の要素を上書きした後、2番目の要素を削除し、要素数を画面に表示している。画面には3が表示される。

```
並び＝配列！123　456　789　作る。
並び！111 書く。
```

```
並び！3 234 上書き。
並び！2　位置で消す。
ラベル！（並び！要素数？）作る。
```

要素の実行

配列に入っているすべての要素に対して、同じ命令を一括して実行させることが可能である。配列に対してブロックをパラメータとして**それぞれ実行**を使うと、配列の要素の数だけブロックが繰り返し実行される。このとき、ブロックのパラメータには配列の各要素が渡される。

次のプログラムでは、画面に「a□,□b□,□c□と書かれた3個のラベルを作り、それらを配列に入れている。3個のラベルには、特に名前は付けていない。続いて、「実行ボタン」という名前のボタンを作り、押したときに「それぞれ実行」で配列の要素の数だけブロックを実行する。ブロックのパラメータである「並び要素」には、実行されるたびに配列の各要素が渡される。その結果、ボタンを押すたびに「並び要素！10 0 移動する」が実行され、画面上の3個のラベルが少しずつ右に移動することになる。

```
並び＝配列！作る。
並び！（ラベル□"a"□作る　0　100　位置）書く。
並び！（ラベル□"b"□作る　0　50　位置）書く。
並び！（ラベル□"c"□作る　0　0　位置）書く。

実行ボタン＝ボタン！"実行"　作る 0 -50 移動する。
実行ボタン：動作＝「
    並び！「｜並び要素｜ 並び要素！10 0 移動する」それぞれ実行。
 □□
```

オブジェクトの親子関係

ドリトルではオブジェクトの間に**親子関係**と呼ばれる関係があり、これが名前（**プロパティ**、変数、**メソッド**）の参照や書き換えと関連している。この規則を理解しておくことは少し複雑なプログラムを作成するときに不可欠である。

「作る」と親子関係

ドリトルのすべてのオブジェクトは親子関係を持っている。あるオブジェクトに対して、親のオブジェクトのことを**プロトタイプオブジェクト**と呼ぶこともある。

親にはその親、そのまた親、…がいる。ドリトルでは、**ルート**（根元という意味）という特別なオブジェクトが1つだけあり、このオブジェクトだけは親を持っていない。それ以外のオブジェクトは直接または間接にルートの子孫である。

さまざまなオブジェクトは何らかのオブジェクトに対するメソッド**作る**の呼び出しによって作り出されるが、このとき、作られたオブジェクトは「作る」を受け取ったメソッドを親として持つようになる。たとえば次の場合□ObjAはObjBの親となる。

```
□□□□□□□□□□作る。
```

子のオブジェクトは親のオブジェクトが持っていたプロパティ（値やメソッド定義）をすべてそのまま引き継ぎ、親と同じように扱うことができる。この仕組みを利用して、タートル、ボタン、配列など、さまざまな標準オブジェクトをプロトタイプとして持つオブジェクトを必要なだけ作って利用できるわけである。

プロパティの参照・書き換えと親子関係

実際には子のオブジェクトは親のオブジェクトのプロパティをコピーして持っているわけではなく、親が誰かだけを覚えている。そして、プロパティを参照しようとするとき、あるプロパティを自分が直接持っていないときは親、そのまた親…のように探して行き、最初に見つかったものを使用する。ルートまで探していっても無い場合は未定義という特別なオブジェクトが結果になる（未定義もルートの子である）。

このため、子を作ってしまった後からでも、親のプロパティを追加したり書き換えたりすると、その結果は子のプロパティとして参照できる。以下の例ではObjBを作った後で追加したプロパティ「身長」が参照できて、「165」が表示される。

```
ObjA=ボタン！作る。
ObjB=ObjA! 作る。
ObjA: 身長＝165。
ラベル! (ObjB: 身長）作る。
```

ただし、上で説明したように、子のオブジェクトに同じ名前のプロパティが見つかった場合はそちらが使われるので、親の同名のプロパティは参照されなくなる。たとえば、次の場合はObjBのプロパティが参照されるので、「180」が表示される。

```
ObjA=ボタン！作る。
ObjB=ObjA! 作る。
ObjB: 身長＝180。
ObjA: 身長＝165。
ラベル! (ObjB: 身長）作る。
```

これにより、「作る」でオブジェクトを生成したあと、必要なところだけプロパティやメソッドを書き換えてカスタマイズし、独自のオブジェクトを作り出せるわけである。

変数とその束縛

変数の束縛とは、プログラムの中に変数名を書いたとき、その変数名がどの「いれもの」を表しているかの対応規則のことである。ドリトルの変数には、グローバル変数、インスタンス変数、ローカル変数の3種類がある。そして、ブロックの中に変数を書いたときどれを意味しているかは、ブロックの使われ方によって変わってくる。これらについて説明する。

グローバル変数

グローバル変数とは、プログラム全体を通じてどこでも参照できるような変数のことをいう。プログラム中のどのブロックの中でもない位置に書いた変数はグローバル変数を意味している。ドリトルでは、グローバル変数とは実はルートのプロパティである。グローバル変数を簡潔に指定できるように、「ルート: x」を単に「: x」と書いてもよい。従って、次の3つの行は（ブロックの中にない場合）どれも同じ意

味である。

```
□□100□
ルート□□□100□
□□□100□
```

タートル、配列などの標準オブジェクトも、単にグローバル変数に初期値として格納されているだけなので、これを書き換えてしまうと、それ以降参照できなくなってしまう。

インスタンス変数

インスタンス変数とは、個々のオブジェクトに付随する情報を保持するための変数のことをいう。ドリトルでは、インスタンス変数は個々のオブジェクトのプロパティに他ならない。ブロックがオブジェクトのプロパティとして格納されていて、メソッドとして呼び出された場合、そのブロックの中の変数で、「¦…¦」の中に書かれているパラメータやローカル変数以外のものは、インスタンス変数を意味している。たとえば次の例では□ObjB:xが21増やされて221になり、それが表示される。

```
□□□□□ボタン！作る。
□□□□□□□200□
□□□□□増加□□|□□□□□□□□□
□□□□□21 増加。
ラベル□□□□□□□□□作る。
```

ここで注意するべきこととして、インスタンス変数の参照もプロパティの参照なので、参照時に見つからないときは親を探しに行くことである。このため、上の例の1行目の「ObjB:x=200□が無くて、グローバル変数xが100の場合、メソッド「増加」を最初に上のように実行したときは□x+d□でグローバル変数が参照されて121 となり、それをプロパティObjB:xに書き込む（新しくインスタンス変数が作られる）ことである。2回目からは、このインスタンス変数が参照されることになる。もしも、参照も更新も常にグローバル変数に対して行いたい場合は、前項で説明したように「：」を付けて次のようにするべきである。

```
□□□□□ボタン！作る。
□□100□
□□□□□増加□□|□□ :□□ :□□□□□
□□□□□21 増加。
ラベル□□□□□□□□□作る。
```

インスタンス変数のうち、自分はオブジェクト自身を示す特別な変数である。

ローカル変数、名前解決規則

ローカル変数とは、ブロック内のコードにだけ関係する概念で、そのブロックの内側でだけ使える変数のことをいう。ブロックのパラメータも、渡された値を初期値に持つという点以外はローカル変数と同等である。ローカル変数は必ず、ブロックの冒頭部分で「¦パラメータ… ; ローカル変数…|」の形で定義される。次のメソッド「距離」では、ブロックの先頭で定義したローカル変数だけを使って2点間の距離を求めている。

```
□□□□□ボタン！作る。
□□□□□距離□□|x1 y1 x2 y2 ; dx dy|
    dx = x1 - x2□dy = y1 - y2□
    sqrt□dx*dx+dy*dy□□□
```

```
□□□□□□□100 100 200 200 距離。
ラベル□□□□作る。
```

ブロックは何重か入れ子になっていることがある。そのとき、内側のブロックで外側のブロックのローカル変数を参照することができる[1)]□

ここまでの説明をまとめると、あるブロックの中に現れる名前が何を表すかは、次のようにして決められている。

1. そのブロック冒頭の ┆…┆ で定義されているものはローカル変数である。
2. そのブロックが直接メソッドとして実行されているなら、それ以外の名前はすべてインスタンス変数である（ただし、参照時は親オブジェクトのプロパティを参照していることもある）。
3. そうでないなら、そのブロックを囲む外側において、同じ規則で名前の使われ方を探索するため1に戻る。
4. 一番外側まで来た場合（どのブロックの中でもなくなった場合）は、名前はグローバル変数である。

字句の約束

プログラミング言語において字句の約束といった場合、名前、数値、文字列などの書き方に関する約束を意味する。ドリトルの場合についての約束を以下で説明していく。

なお、以下では込み入った書き方の規則を表すのに拡張BNFと呼ばれる記法を用いている。その記法と意味は次の通り:

X ::= α	書き方αをXと呼ぶ、ないしXの定義。
α□β	αまたはβ□
[α]	αまたは空。
α…	αが1個以上並んだもの。
(α)	αとおなじ（かっこによりまとめる）。

空白と改行

ドリトルは自由書式（フリーフォーマット）の言語である。つまり、プログラムの配置は見やすさなどを考えて空白や改行の配置を比較的自由に選ぶことができる。規則は次の通り。

- プログラムは何行に渡ってもよい。行の切れ目は空白を入れられる位置であれば、どこにでも入れられる。
- 名前の途中、数値の途中には空白を入れられない。
- 名前と名前、名前と数値、数値と数値が連続しているところには、空白（または改行）を入れなければならない。
- 空白は何個あっても1個と同じ意味になる（文字列の中を除く）。

たとえば、次の2つのプログラムは同じ動作になる。

```
ボタン1 = ボタン!作る。
```

```
ボタン1
    =
  ボタン!作る。
```

また、次のプログラムは名前と数値がくっついているため正しくない。（実行するとエラーになる）

```
ボタン1 = ボタン!"テスト"作る100 50位置。
```

名前

ドリトルの名前（識別子）は変数名やプロパティ名などを指定するのに使われる。その規則は次の通り。

```
名前文字 ::= 漢字 | ひらかな | カタカナ | アルファベット
名前 ::= 名前文字 [ 名前文字 | 数字 ] …
```

すなわちドリトルの名前は「漢字、ひらかな、カタカナ、アルファベットのいずれかではじまり、その後に、漢字、ひらかな、カタカナ、アルファベット、数字が任意個続いたもの」である。名前の正しい例と正しくない例を示す:

```
○ ボタン1　X　A1
× ボタン-1　1番目
```

数値

ドリトルの数値はプログラム上に定数（数オブジェクト）を直接記述するのに用いる。その規則は次の通り。

```
符号 ::= ＋ | －
数値 ::= [ 符号 ] 数字… [ . 数字… ] [ 名前文字… ]
```

すなわち、数値は「先頭に＋または－の符号があってもなくてもよく、その後に数字が1個以上続き、その後に小数点と1個以上の数字があってもなくてもよく、その後に1個以上の名前文字があってもなくてもよい」ものである。最後の名前文字は数値の値としては無視されるが、読みやすさのために書くことができる。数値の正しい例と正しくない例を示す:

```
○ 1 +55 -3.1416 3番目
× 15. 22時59分
```

文字列

ドリトルの文字列は、文字列オブジェクト（文字のならびを表現するオブジェクト）の定数をプログラム中に直接書くのに用いる。その規則は次の通り。

```
文字列 ::= " [ 文字… ] " | 『 [ 文字… ] 』
```

文字列の正しい例と正しくない例を示す:

```
○ "あいう"　□ABC□
× "あい　□ABC
```

大文字と小文字/全角と半角

ドリトルのプログラムでは、大文字と小文字、16ビット文字（いわゆる全角）と8ビット文字（いわゆる半角）を区別しない。したがって次の各行の名前はいずれも同じものとして扱う。

```
ABC  Abc  □□C
```

このほか、「＋」「！」などの記号類についても同様である。ただし、文字列の中だけは例外であり、大文字と小文字、16ビット文字と8ビット文字の違いも含めて中に書かれた文字の並びをそのまま表す。

このほか、いくつかの日本語記号と英語記号を同等に扱う。小数点および文の終りを表すのには「。」と「.」のいずれを使ってもよい。メッセージ送信を表すのに「！」と「、」のいずれを使ってもよい。そして、**ブロック**を表すのに「…」と[…]のいずれを使ってもよい。

たとえば、次のドリトルの文はいずれも同じ意味になる。

```
ボタン1＝ボタン！作る。
ボタン1:プロパティ ＝ □x ＝ x ＋ 10□ボタン1!□x□ 50 位置」。
ボタン1：プロパティ ＝ [□ □ □ ＋ 10. ボタン1□□□□ 50 位置].
```

コメント

「/ /」から行末までは注釈□コメント）と解釈され、プログラムの一部として扱われない。人が読むためのメモや、一時的にプログラムの一部を実行したくないときのコメントアウトなどに使うことができる。

たとえば、次の例で、1行目はプログラム全体の注釈、2行目はその処理の注釈、3行目は実行されないように一時的にプログラムの一部をコメントアウトしている。

```
// タートルのテストプログラム
かめた＝タートル！作る。 // かめたを作る
// かめきち＝タートル！作る。
```

[1)]
内側のブロックをグローバル変数などに保管しておいて、後で（外側のブロックの実行が終了してしまってから）実行した場合の動作は未定義である。

From:
https://dolittle.eplang.jp/ - **プログラミング言語「ドリトル」**

Permanent link:
https://dolittle.eplang.jp/ch_syntax?rev=1594412115

Last update: **2020/07/11 05:15**