

標準オブジェクト

ドリトルには、さまざまな種類のオブジェクトが標準で用意されている。ここでは、ドリトルのプログラムを作るときに、さまざまなプログラムで共通に使われるオブジェクトについて概要を説明する。ブロック、タイマー、配列については「[ドリトル言語の基礎知識](#)」を参照されたい。ネットワーク、ロボットなど外部機器制御、音楽演奏などのオブジェクトについては、本文の各章と「[レファレンス](#)」を参照されたい。

数値オブジェクト

数値オブジェクトは、数を表すオブジェクトである。数値オブジェクトは、1や10.0のような数値定数を書くことによって作り出されることもあるし、さまざまな演算の結果として作り出されることもある。数値定数は、0b1100のような2進表現や0xFFのような16進表現でも記述できる。

数値に対する演算としては、四則演算「+」「-」「*」「/」と剰余の演算「%」を使うことができる（積は「*」を、商は「/」を使うこともできる）。また、比較演算子「==」「!=」

「>」、「>=」、「<」、「<=」も使うことができる。このほか、関数として、平方根 `sqrt`、三角関数 `sin`、`cos`、`tan`、四捨五入 `round`、絶対値 `abs` などがある。

数値を扱う特別な関数として、乱数がある。乱数は、実行するたびに異なる数を返す関数である。たとえば、ゲームで実行するたびに少しずつ違う動きをさせたいときに使うと便利である。乱数を使うときは、「乱数(3)」のように数値を指定する。この例のように正の整数を指定したときは、1からその数までの間の数をランダムに返す。次のプログラムを実行すると、実行するたびに1から3の数字がランダムに表示される。

```
ラベル! (乱数(3)) 作る。
```

負の数を含む乱数を発生させたいときは、発生させたい数の幅を指定して乱数を生成し、その半分より1だけ大きい値を引くことで、0を中心とした正負の乱数を生成することができる。次のプログラムを実行すると、-5から5の範囲の数字がランダムに表示される。

```
ラベル! (乱数(11) - 6) 作る。
```

数値にメソッドを定義することで、すべての数値オブジェクトからそのメソッドを利用できる。次のプログラムを実行すると、その数の2倍の数が表示される。

```
数値: 二倍 = 「自分 * 2」。  
ラベル! (3! 二倍) 作る。
```

括弧で囲まれた数式の中では、パラメータを使わないメソッドを関数の形で使うことができる。

```
数値: 二倍 = 「自分 * 2」。  
ラベル! (二倍(3)) 作る。
```

数値オブジェクトは、内部では64ビット倍精度浮動小数点数として扱われる。浮動小数点数は精度の高い近似値であるが誤差を含むため、整数値の扱いに関して次の補正を行っている。¹⁾

- 表示などで文字列に変換する際に、最も近い整数値との差分が十分に小さい場合は整数値として

出力する。

- 値が等しい (=) ことを比較する際に、最も近い整数値との差分が十分に小さい場合は等しいと判定する。

多倍長整数オブジェクト

多倍長整数オブジェクトは、大きな整数を表すオブジェクトである。数値オブジェクトは有効数字が17桁程度であるため、数十桁以上の巨大な整数値を扱うときは多倍長整数を使用する。多倍長整数の値は、次の例のように、文字列に**大きい整数にする**を送り生成する。

```
x "1000000" 大きい整数にする。
```

数値オブジェクトの値から多倍長整数の演算を行う場合には、事前に多倍長整数に変換しておく必要がある。次の例はどちらも2の70乗を計算している。正しい値は「1180591620717411303424」である。ところがxは2という数値オブジェクトのまま計算しているため、値は「1180591620717411300000」となり、誤差が生じる。yは2から多倍長整数を生成してから計算しているため、すべての桁が正しく計算される。

```
x=2 70 pow
y=(2 大きい整数にする) 70 pow
ラベル x) 作る。
ラベル y) 作る 次の行。
```

多倍長整数に対する演算としては、四則演算「+」「-」「*」「/」と剰余の演算「%」を使うことができる（積は「x」を、商は「÷」を使うこともできる）。また、比較演算子「==」「!=」

「>」「>=」「<」「<=」も使うことができる。このほか、関数として、絶対値「abs」、べき乗（「pow」などがある）

文字列オブジェクト

文字列オブジェクトは、0個以上の文字の並びを表すオブジェクトであり、“こんにちは”や“abcxyz”のように、プログラムの中に文字列定数を書いて生成することもあればGUI部品やファイルから読み込むことで作られることもある。

文字列は、「+」で連結することができる。次のプログラムを実行すると、2つの文字列が連結されて、画面に「こんにちはかめたさん」という文字列が表示される。

```
x "こんにちは"。
y "かめたさん"。
ラベル x y) 作る。
```

また、文字列オブジェクトの中身が“123.45”などのように数値定数の形をしているときは、「+」も含めて数値として演算したり比較される。したがって、次の例では結果は「14」になる。

```
x "3"
y "11"
ラベル x y) 作る。
```

部分を使うと、文字列の一部を取り出すことができる。パラメータに1個の数字（たとえばmと呼ぶ）を指定したときは、先頭からm番目以降の文字が返される。たとえば、4を指定すると、4文字目以降の文字を返す。次のプログラムを実行すると、「こんにちは」という文字列の4文字目以降が切り出されて、画面に「ちは」という文字列が表示される。

```
x["こんにちは"]  
ラベル[x[4 部分)]作る。
```

パラメータに2個の数字（たとえばm, nと呼ぶ）を指定したときは、先頭からm番目の文字からn個の文字が返される。たとえば、2と3を指定すると、2文字目から3文字を返す。次のプログラムを実行すると、「こんにちは」という文字列の2文字目から3文字が切り出されて、画面に「んにち」という文字列が表示される。

```
x["こんにちは"]  
ラベル[x[2 3 部分)]作る。
```

含む?を使うと、ある文字列の一部に別の文字列が含まれているかどうか調べることができる。

```
x["上山田"]  
[x["山"含む?]]!なら「ラベル! "こんにちは" 作る」実行。
```

文字列にメソッドを定義することで、すべての文字列オブジェクトからそのメソッドを利用できる。次のプログラムを実行すると、その文字列が2回続いた文字列が表示される。

```
文字列: 二倍 = 「自分 + 自分」。  
ラベル! ("こんにちは" ! 二倍) 作る。
```

括弧で囲まれた数式の中では、パラメータを使わないメソッドを関数の形で使うことができる。

```
文字列: 二倍 = 「自分 + 自分」。  
ラベル! (二倍 ("こんにちは")) 作る。
```

真偽値

真偽値とは、条件が成り立つ/成り立たないなど、真偽の区別を表す値である。比較演算子や成否を調べるメソッドはどれも真偽値を返す。

ドリトルでは真偽値とは、真を表すオブジェクトと偽を表すオブジェクトがそれぞれ1つずつあり、そのどちらかであるという意味になる。また、グローバル変数「真[はい)」と「偽[いいえ)」にそれぞれ真と偽を表すオブジェクトが格納してある。たとえばxがyよりもzよりも大きいかどうかを調べる例を示す。

```
x=7 y=5 z=3  
xが最大 = 真。  
[x < y]なら「xが最大 = 偽」実行。  
[x < z]なら「xが最大 = 偽」実行。  
[xが最大]!なら「ラベル「xが最大です」作る」実行。
```

メソッド反対は、真偽値を反転する（**NOT**）。つまり真なら偽、偽なら真を返すのに使える。また、複数の真偽値がすべて成り立つかどうか（**AND**）を調べるには**ぜんぶ**の、複数の真偽値のどれか1つ以上が成り立つかどうか（**OR**）を調べるには**どれか**の、メソッド本当を使うことができる。たとえば上の

例は次のように書ける。

```
x=7□y=5□z=3□  
「ぜんぶ□x>=y□x>=z□本当」！なら  
「ラベル□"xが最大です" 作る」実行。
```

メソッド「本当」に与えられたパラメータがブロックの場合には、そのパラメータは必要になったときだけ実行される。次のプログラムでは、最初に「x>=y」が実行される。その値が真の場合には両方の条件が真であるかを調べるために続く「x>=z」が実行されるが、値が偽の場合には結果がその時点で決まってしまうため続く「x>=z」は実行されない。

```
x=7□y=5□z=3□  
「ぜんぶ□x>=y□x>=z□本当」！なら  
「ラベル□"xが最大です" 作る」実行。
```

色オブジェクト

色オブジェクトは、さまざまな色を表すオブジェクトである。よく使う色の色オブジェクトが、その色の名前のグローバル変数に格納してある。具体的には、「黒□赤□緑□青□黄色□紫□水色□白」の8色が用意されている。これらを光と絵具のパレットオブジェクトで混ぜ合わせて新しい色を作ることができる。

任意の色を作る場合には、赤、緑、青の強さを0～255の整数で指定して色オブジェクトを作る。各色を16進の2桁ずつの数値で指定することも可能である。

```
新しい色 = 色 ! 200 150 255 作る。  
新しい色 = 色 □ 0xC896FF 作る。
```

タートルオブジェクト

タートルオブジェクトは、画面上でグラフィックスを用いて絵を描いたり、任意の画像に「変身」させてそれを画面上に置いたり動かしたりするのに使うオブジェクトである。作り出すには、タートルの「作る」を使う。

作った状態では、動くとき軌跡が残る状態になっているが、動いても軌跡が残らない状態に切り替えることもできる。切り替えにはメソッド「ペンあり」「ペンなし」を呼び出せばよい。

移動を制御するには、現在の位置と向きを基準として前進や後退を指定する方法（タートルグラフィックス）が多く使われる。これには、動く長さを指定した「歩く」「戻る」、回転角度を指定した「右回り」「左回り」が使える。移動量は画面上のピクセル単位、回転量は度数で指定する。

このほか□X方向とY方向の移動量を指定して現在位置を起点にその分だけ移動する「移動する」、絶対座標（画面中心が原点）でX座標とY座標を指定してその座標位置に移動する「位置」も使える。一連の軌跡を描き始めた位置に戻る（軌跡を閉じた図形する）には「閉じる」が使える。

軌跡は閉じていてもいなくても、「図形を作る」で切り離して独立した図形オブジェクトにできる（図形オブジェクトについては次項）。

タートルの現在の位置や向きは「縦の位置？」「横の位置？」「向き？」で取得できる。また、「線の色」「線の太さ」で軌跡の色や太さを変更できる。

タートルの画像は最初は亀の絵だが、これを任意の画像に変更できる。それには、画像ファイル名の文

字列を指定して「変身する」を呼ぶ。また、画像の表示をON/OFFもできる。これには「消える」と「現れる」を使う。

タートルオブジェクトの重要なメソッドとして「衝突」がある。これは特定のメソッドがあるのではなく、プログラムを作る人が「衝突」という名前のメソッドを定義することで、タートルが動いていって他の図形やタートルと接触したときにこのメソッドが呼び出されるようにできる。たとえば、タイマーを使って一定ペースで前進するタートルに衝突メソッドで「数歩下がって90度向きを変える」ような動作を定義しておけば、前進していき何かにつつかるとよけるような動作を行うようになる。「衝突」が呼び出されるときには、パラメータとして衝突した相手のオブジェクトが渡されるので、それに対して何かの作用を施すこともできる。

タートルにメソッドを定義することで、すべてのタートルオブジェクトからそのメソッドを利用できる。次のプログラムを実行すると、すべてのタートルが何かに衝突したときに斜め後ろに跳ね返るようになる。

```
タートル：衝突 = 「自分！150 右回り」。  
かめた = タートル！作る。  
かめきち = タートル！作る 100 0 位置。  
タイマー！作る「かめた！10 歩く」実行。
```

図形オブジェクト

図形オブジェクトは、タートルによって描いた図形を切り離したものであり、完結した1つのオブジェクトとして色を塗ったり動かしたりできる。

動かし方や見え方として、「右回り」「左回り」「移動する」「位置」「消える」「現れる」はタートルと同じに使うことができる。

また、倍率を指定して「拡大する」を呼ぶことで大きさを拡大/縮小できる。倍率を2つ指定することもでき、そのときは縦と横の倍率を別々に指定したことになる。色を指定して「塗る」を呼ぶことで色を付けることもできる。

図形オブジェクトにもタートルオブジェクトと同様に「衝突」メソッドを定義することで、他の図形やタートルとの接触を検知することができる。

図形にメソッドを定義することで、すべての図形オブジェクトでそのメソッドを利用できる。次のプログラムを実行すると、すべての図形に横幅が広い形になる「変形」というメソッドが定義される。

```
図形：変形 = 「自分！21 拡大する」。  
かめた = タートル！作る。  
さんかく = 「かめた！100 歩く 120 左回り」！3 繰り返す(赤) 図形を作る。  
さんかく！変形。
```

GUI部品

GUI部品とは、ボタンやスライダーなど、画面上に現れてユーザのマウス操作などによって動作させられるようなものをいう。どれも、画面上の配置や表示を制御するのに「位置」「移動する」「消える」「現れる」を(タートルと同様に)使うことができる。また、縦方向と横方向の大きさ(ピクセル単位)を指定して「大きさ」を呼び出すことで大きさを設定できる。また、色を指定して「塗る」で地の色、「文字色」で文字の色を設定できる。GUI部品は画面に表示されるが、タートルや図形と重なっても衝突は起こらない。以下で、個々のGUI部品オブジェクトについて説明する。

ラベル

ラベルは長方形の領域で、その上に文字列を表示することができる。表示する文字列は**作る**で生成するときにパラメータとして指定する。また、文字列を指定して「書く」を呼ぶことで文字を取り替えることもできる。次のプログラムは、画面に変数の値を表示する。実行すると、10が表示される。

```
x□10□  
ラベル□□x□作る。
```

ラベルなどいくつかのオブジェクトでは、表示する文字列の中にHTMLを記述できる。文字列の先頭は必ず“ □html□”で始まる必要がある。

```
x□"<html><p color=blue>こんにちは</p></html>"□  
ラベル□□x□作る。
```

ボタン

ボタンはラベルと見た目は似ていて、「作る」で文字列を指定することも同じだが、さらに**動作**という名前のメソッドを定義しておくことで、ボタンが押したときにそのメソッドが呼び出される。たとえば、次のプログラムを実行すると、画面にボタンが表示され、ボタンをクリックするたびにボタンは少しずつ右に移動する。

```
ボタン1 = ボタン！"テスト" 作る。  
ボタン1：動作 = 「自分！10 0 移動する」。
```

フィールド

フィールドもラベルと見た目は似ているが、文字列が表示できるだけでなく、そこにユーザが文字列を入力することができる。「作る」や「書く」で文字列を設定できることは同じだが、加えて「読む」で現在入っている文字列を取り出すことができる。次のプログラムを実行すると、ボタンを押したときにテキストフィールドに入力された文字を読み取り、かめたは指定された歩数だけ前進する。

```
かめた = タートル！作る。  
窓 = フィールド！作る。  
前進ボタン = ボタン！"前進" 作る。  
前進ボタン：動作 = 「かめた！（窓！読む） 歩く」。
```

フィールドでは、**リターンキー**を押すことで、ボタンを押したときのように入力完了を受け取ることができ、定義しておいたメソッド「動作」を呼び出させることができる。次のプログラムを実行すると、リターンキーを押したときにテキストフィールドに入力された文字を読み取り、かめたは指定された歩数だけ前進する。

```
かめた = タートル！作る。  
窓 = フィールド！作る。  
窓：動作 = 「かめた！（自分！読む） 歩く」。
```

スライダー

スライダーは、マウスで中のレバーを動かして値を指定できるようなオブジェクトである。配置を指定するのに、「縦向き」「横向き」を呼ぶことでスライダーの向きを設定できる（指定しないと縦向き）。

また、「文字出す」「文字消す」で目盛りの表示をON/OFFできる。

バーが動くと動作メソッドが実行され、そのときの値がパラメータとして渡される。値の範囲は0～100である。次のプログラムを実行すると、レバーを動かすたびに位置の値が表示される。

```
バー = スライダー! 作る。  
結果 = ラベル! 作る。  
バー: 動作□□□x□□結果□□x□書く」。
```

選択メニュー

選択メニューは、ユーザが複数の候補から選択するメニューを作り出すオブジェクトである。選択肢は作るのパラメータとして指定する。

選択肢が選ばれると動作メソッドが実行され、選ばれた選択肢がパラメータとして渡される。次のプログラムを実行すると、メニューで選択した候補が表示される。

```
メニュー1 = 選択メニュー! "最初" "次" "最後" 作る。  
ラベル1 = ラベル! 作る。  
メニュー1: 動作□□|x| ラベル1!(x)書く」。
```

リスト

リストは複数の文字列を表示するオブジェクトである。文字列を指定してメソッド「書く」を呼ぶたびに、その文字列が内容に追加されていく。また、番号を指定して「読む」を呼ぶと、その番号の行の文字列を取得できる。次のプログラムを実行すると、画面に「こんにちは!」という文字と「いいお天気ですね」という文字が表示される。

```
窓 = リスト! 作る。  
窓! "こんにちは!" 書く。  
窓! "いいお天気ですね" 書く。
```

オブジェクトの保存と読み出し

オブジェクトの保存

オブジェクトファイルオブジェクトを使うと、オブジェクトをコンピュータにファイルとして保存しておき、再び取り出して使うことができる。たとえば、ゲームの得点などを保存しておけば、次回の実行時に「今までの最高得点」などを表示することが可能になる。保存できるオブジェクトは、数値□文字列□配列である。

オブジェクトファイルを使うときは、ファイル名を指定して作るでオブジェクトファイルを作った後で、オブジェクトを読み書きする。次のプログラムでは□□objfile.txt□という名前のファイル名を指定してオブジェクトファイルを作った後で、「得点」、「名前」、「友人」という数値、文字列、配列のオブジェクトを、それぞれ□point□□name□□friends□という名前を付けて書くで保存している。

保存するときの名前は、自由に付けて構わない。ここでは「得点」を「point□という英語の名前で保存したが、「得点」という同じ名前で保存することもできる。

```
得点 = 10。
```

```
名前 = "かめた"。  
友人 = 配列！"かめきち" "かめこ" 作る。
```

```
ファイル = オブジェクトファイル "objfile.txt" 作る。  
ファイル "point" (得点) 書く。  
ファイル "name" (名前) 書く。  
ファイル "friends" (友人) 書く。
```

オブジェクトの読み出し

オブジェクトを読み込むときは、保存したときの名前を指定して読み出せばよい。次のプログラムでは `objfile.txt` という名前のファイル名を指定してオブジェクトファイルを作った後で、`point` `name` `friends` という名前のオブジェクトを取り出している。取り出したオブジェクトは、好きな名前を付けて使うことができる。

```
ファイル = オブジェクトファイル "objfile.txt" 作る。  
点数 = ファイル "point" 読む。  
氏名 = ファイル "name" 読む。  
友だち = ファイル "friends" 読む。
```

```
リスト！作る（点数）書く（氏名）書く（友だち）書く。
```

オブジェクトの削除

オブジェクトを削除するときは、保存したときの名前を指定して削除すればよい。次のプログラムでは `objfile.txt` という名前のファイル名を指定してオブジェクトファイルを作った後で、`point` という名前のオブジェクトを削除している。オブジェクトを読み込んで表示すると `point` というオブジェクトの値は、「値が存在しない」という意味を表す未定義オブジェクトである「**[undef]**」が表示される。

```
ファイル = オブジェクトファイル "objfile.txt" 作る。  
ファイル "point" 消す。  
点数 = ファイル "point" 読む。  
氏名 = ファイル "name" 読む。  
友だち = ファイル "friends" 読む。
```

```
リスト！作る（点数）書く（氏名）書く（友だち）書く。
```

テキストファイルの操作

1行ごとの追加書き込み

テキストファイルオブジェクトを使うと、文字列をファイルに読み書きできる。テキストファイルを使うときは、ファイル名を指定して作るでオブジェクトを作った後で、読み書きする。次のプログラムでは `textfile.txt` という名前のファイル名を指定してオブジェクトを作った後で、「メッセージ1」と「メッセージ2」の変数の文字列を書き込んでいる。

```
メッセージ1 = "こんにちは"。  
メッセージ2 = "かめたです"。
```

```
ファイル = テキストファイル "textfile.txt" 作る。  
ファイル! (メッセージ1) 書く。  
ファイル! (メッセージ2) 書く。
```

ファイル全体の書き込み

配列に入っている文字列を、テキストファイルに書き込むことができる。次のプログラムでは `[]textfile.txt` という名前のファイル名を指定してオブジェクトを作った後で、「内容」という名前の配列を書き込んでいる。

```
内容 = 配列! "こんばんは" "かめきちです" 作る。
```

```
ファイル = テキストファイル "textfile.txt" 作る。  
ファイル! (内容) 全部書く。
```

ファイル全体の読み出し

テキストファイルの内容は、配列に読み込んで使うことができる。次のプログラムでは `[]textfile.txt` という名前のファイル名を指定してオブジェクトを作った後で、ファイルの内容を「内容」という名前の配列に読み込んでいる。

```
ファイル = テキストファイル "textfile.txt" 作る。  
内容 = ファイル! 読む。
```

```
ラベル! (内容) 作る。
```

1)

「十分に小さい」差分の値は、V2.33では「0.0000001」を使用している。

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_common_object?rev=1518142688

Last update: **2018/02/09 11:18**

