

# はじめに

## 本書について

この本では、ドリトルというプログラミング言語を用いて、ITの本質であるプログラミングを分かりやすく学べるように解説しています。ドリトルは初心者でもプログラミングを学びやすいように設計された教育用言語で2000年に筑波大学の久野靖教授と筆者によって設計されました。教育用でありながら、現在主流のオブジェクト指向の考え方を基礎にしていることが特徴です。

ドリトルの特徴を手早く知りたい方は、Activity 5の宝物拾いゲームを、1行ずつプログラムを入力しながら実行してみてください。説明するときのシナリオも用意されており、中学校や高校を中心に、多くの授業で利用されています。<sup>\*1</sup>

本書では、ドリトルで作成可能な「グラフィックス（アニメーション）」「ゲーム」「音楽演奏」「ネットワーク通信」「外部機器制御」という5種類のプログラミングを解説しています。興味に応じて、必要な章を参照してください。

- Part I「ドリトルを使ってみよう」は、必ず最初にお読みください。
- Part II「絵を描こう」は、プログラムで絵を描いたりアニメーション作品を作ります。
- Part III「ゲームを作ろう」は、応用として、いくつかのゲーム作品を作ります。
- Part IV「音楽を演奏しよう」は、プログラムで音楽を演奏します。
- Part IX「ネットワークで通信しよう」は、通信するプログラムを作ります。
- Part VIII「ロボットを動かそう」は、外部機器を制御するプログラムを作ります。
- Part VI「センサやLEDと入出力しよう」は、計測制御を行うプログラムを作ります。

付録には、言語の解説や命令の一覧が掲載されています。必要に応じて参照してください。

- 付録A「授業での利用」では、小学校から大学までの授業例が紹介されています。
- 付録B「よいプログラムを書くために」では、プログラムを書くときの基本的な考え方が解説されています。
- 付録C「ドリトル言語の基礎知識」と付録D「標準オブジェクト」では、ドリトルの

---

<sup>\*1</sup> 詳しくはドリトルのサイトを参照。<http://dolittle.eplang.jp/>

文法と、多くのプログラムで共通に使われる標準オブジェクトを解説しています。

- 付録 E「ドリトルの命令一覧」では、ドリトルのプログラムで使える命令をオブジェクトごとに解説しています。

詳しい情報は、Web サイトをご覧ください。授業等の連絡は筆者にお願いします。

- ドリトルの Web サイト: <http://dolittle.eplang.jp>
- 筆者（兼宗）の連絡先: [kanemune@acm.org](mailto:kanemune@acm.org)

## 協力していただいた方々

本書の作成にあたっては、多くの方々の協力をいただきました。

共著者である筑波大学の久野靖先生には、本書の付録 B「よいプログラムを書くために」と付録 C「ドリトル言語の基礎知識」の執筆を担当していただきました。

外部機器制御機能については、静岡大学の紅林秀治先生、愛知教育大学の鎌田敏之先生、神戸市立科学技術高校の中野由章先生、イーテキスト研究所の青木浩幸氏、スタジオミュウの井上修次氏、大阪電気通信大学の大村基将先生、株式会社アーテックに協力をいただきました。音楽機能については、東京農工大学の辰己丈夫先生、クジラ飛行機こと酒徳峰章氏、イーテキスト研究所の山澤昭彦氏、大阪府立桃谷高校の野部緑先生に協力をいただきました。言語の検討に関しては、東京農工大学の並木美太郎先生に協力をいただきました。

英語、韓国語、中国語をはじめとする多言語機能については、高麗大学の李元揆研究室、長野大学の和田勉先生、明星大学の長慎也先生、京都情報大学院大学の江見圭司先生をはじめとする方々に協力をいただきました。

ドリトルの教育利用については、松阪市立飯高西中学校の井戸坂幸男先生、神奈川県立柏陽高校の間辺広樹先生、神奈川県立相模向陽館高校の保福やよい先生、北海道小樽潮陵高校の佐々木寛先生、大阪電気通信大学の島袋舞子さんおよび、情報処理学会コンピュータと教育研究会（CE 研）、大阪電気通信大学兼宗研究室のメンバーをはじめとする方々に多くの示唆をいただきました。

TeX による組版については、三重大大学の奥村晴彦先生に全面的な協力をいただきました。イーテキスト研究所の原久太郎氏には、出版に関して多大な尽力をいただきました。

これら、ドリトルとアンブラグドの活動を支えてくださっている、多くの仲間、友人、家族に心から感謝いたします。

2015 年 5 月 兼宗 進

# ドリトルのパッケージ

本書の執筆時点では、次の環境で動作を確認している。<sup>\*2</sup>

- OS：Windows（7/8）、Mac OS X（10.10 以降）、Linux
- Java：1.8 以降

ドリトルは、パソコンに**ローカル版**をインストールして動かせるほか、Web ブラウザで**オンライン版**を動かすことも可能である。ローカル版は、コンピュータにインストールする必要があるが、機能の制限はない。オンライン版は、コンピュータにソフトをインストールするのが難しい場合などに便利であるが、プログラムをディスクに保存することができず、一部の機能に対応していないなどの制限がある。表 1 にローカル版とオンライン版の機能を示す。

表 1 ローカル版とオンライン版

機能	ローカル版	オンライン版
インストール	△必要（※1）	○不要
プログラム保存	○可能	△不可（※2）
グラフィックス	○可能	○可能
音楽	○可能	○可能
ネットワーク通信	○可能	×不可
外部機器制御	○可能	×不可

- （※1）ファイル展開のみであり、管理者権限でのインストールは不要。  
（※2）コピー＆ペーストにより保存が可能。

<sup>\*2</sup> これ以外の環境でも、JRE（Java Runtime Environment）があれば実行することが可能である。

## ローカル版をインストールして動かす

ローカル版は、ドリトルのサイト (<http://dolittle.eplang.jp>) のダウンロードページから最新のバージョンを入手できる。インストール手順は、ダウンロードページのバージョンごとの説明を参照されたい。

### Windows、Linux の手順

- Java をインストールしておく。(Java 同梱版では不要)
- ドリトルのサイト (<http://dolittle.eplang.jp>) から zip ファイルをダウンロードする。
- ダウンロードしたファイルを展開する。zip ファイルの展開は、インターネットなどで公開されている各種のフリーソフトを利用できる。
- 作成されたフォルダを適切なディレクトリ (Windows では「C:¥Program Files」など) に移す。
- フォルダの中にある起動ファイル (Windows では `dolittle.exe` または `dolittle.bat`、それ以外の OS では `dolittle.sh`) をクリックしてドリトルを起動する。

### Macintosh の手順

- Java をインストールしておく。
- ドリトルのサイト (<http://dolittle.eplang.jp>) から **Macintosh 版** をダウンロードする。
- ダウンロードしたファイルをクリックし、開いたウィンドウで「Dolittle」のアイコンをマウスでドラッグして「アプリケーション」のアイコンにコピーする。
- 必要に応じて、「アプリケーション」フォルダの「Dolittle」を「Dock」(画面の下または横にあるアイコンのバー) にドラッグして登録する。
- MYU や Arduino などの外部機器を制御する場合は、事前にターミナルから管理者権限で次のコマンドを実行する。

```
sudo mkdir /var/lock
```

```
sudo chmod 777 /var/lock
```

- V2.2 以降のドリトルでは、`startup.ini` などの初期化ファイルや、プログラム中で利用する画像ファイルは、ユーザーごとの「書類」フォルダにある「Dolittle」フォルダに置いて使用する。<sup>\*3</sup>

---

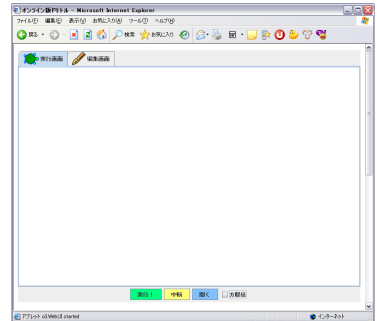
<sup>\*3</sup> フォルダは初回にドリトルを起動したときに作成される。V2.38 からフォルダの場所が「ライブラリ/Dolittle」から「書類/Dolittle」に変更された。

## オンライン版を Web ブラウザで動かす

### オンライン版を表示する

Google Chrome、Safari、Internet Explorer、Firefox、Opera などの **Web ブラウザ** から、**ドリトルの Web サイト** (<http://dolittle.eplang.jp>) にアクセスし、「オンライン版」というリンクをクリックする。すると、右の図のような画面が表示される。

ドリトルの画面には上部に**タブ**があり、「実行」「編集」を切り替えられる。プログラムは**編集画面**に入力する。画面下の「実行！」と書かれた**実行ボタン**を押すとプログラムの実行が行われ、**実行画面**に結果が表示される。



### プログラムの保存

オンライン版のドリトルではプログラムを保存することができない。残しておきたいプログラムは、**コピー&ペースト**（コピー／貼り付け）などの方法でテキストエディタ（エディタ）やワードプロセッサ（ワープロ）などのアプリケーションソフトに移してからファイルに保存する必要がある。

## 書籍（第2版）からの主な修正点

### ドリトルの変更点（第2版のV2.2からの変更点）

- Windowsで「dolittle.bat」と「dolittle.exe」から起動できるようになった。  
(p.i)
- WindowsでJavaを同梱したパッケージが使えるようになった。(p.i)
- Macのプログラムフォルダが「ライブラリ/Dolittle」から「書類/Dolittle」に変更された。(p.i)
- 編集画面にフォントサイズ変更のショートカットキー（Ctrl-↑, Ctrl-↓, Ctrl-0）が追加された。(p.2)
- LeapMotionに対応した。(p.64)
- Studuinoに対応した。(p.78)
- ロボット制御の構文が変更された。画面でのシミュレーションが可能になった。  
Windows(64bit)に対応した。(p.92)
- Arduinoに書き込むプログラムファイルの拡張子が「ino」に変更された。(p.68)
- 文字列で「文字コード」が使えるようになった。(p.182)
- 「停止」でそのタイマーのすべての実行が停止されるようになった。(p.186)
- 要素数より大きな位置の「上書き」で配列が拡張されるようになった。(p.188)
- 配列に「連結」、「加工」、「最大」、「最小」が追加された。(p.188)
- システムに表示、確認、入力、選択のダイアログが追加された。(p.194)
- 画面の「背景画像」で背景の画像を表示できるようになった。(p.199, p.204)
- 「組図形」オブジェクトで複数の図形をまとめて操作できるようになった。(p.205)
- フィールドにフォーカスが追加された。(p.213)
- リストに配列の代入が追加された。(p.214)

# 目次

<b>Part I</b>	<b>ドリトルを使ってみよう</b>	<b>1</b>
<b>Activity 1</b>	<b>はじめてのプログラミング</b>	<b>2</b>
1.1	オブジェクトを作る . . . . .	2
1.2	オブジェクトに名前を付ける . . . . .	3
1.3	オブジェクトに命令する . . . . .	4
1.4	命令を順に実行する (1) . . . . .	4
1.5	命令を順に実行する (2) . . . . .	5
1.6	繰り返し . . . . .	7
1.7	命令の定義 . . . . .	8
	●プログラムのデバッグ . . . . .	10
<b>Part II</b>	<b>絵を描こう</b>	<b>13</b>
<b>Activity 2</b>	<b>描いた絵に色を塗ろう</b>	<b>14</b>
2.1	図形の生成 . . . . .	14
2.2	図形の複製 . . . . .	15
2.3	色オブジェクト . . . . .	16
<b>Activity 3</b>	<b>ペイントソフトを作ろう</b>	<b>18</b>
3.1	作成するペイントソフト . . . . .	18
3.2	ボタンの生成 . . . . .	19
3.3	動作の定義 . . . . .	19
3.4	タートルを操作する . . . . .	20
3.5	図形を描く . . . . .	21
3.6	図形に色を塗る . . . . .	22
<b>Activity 4</b>	<b>アニメーション</b>	<b>23</b>
4.1	今まで学んだ繰り返し . . . . .	23

4.2	タイマーオブジェクト . . . . .	23
4.3	複数のオブジェクトを動かす . . . . .	25
4.4	タイマーで複数の実行を行う . . . . .	26
4.5	タイマーの終了を待つ . . . . .	26

## Part III ゲームを作ろう 29

### Activity 5 宝物拾いゲーム 30

5.1	作成するゲーム . . . . .	30
5.2	タートルを操作する (ステップ 1) . . . . .	31
5.3	タートルを前進させる (ステップ 2) . . . . .	31
5.4	宝物を画面に置く (ステップ 3) . . . . .	31
5.5	宝物を拾う (ステップ 4) . . . . .	32

### Activity 6 ピンポンゲーム 34

6.1	作成するゲーム . . . . .	34
6.2	壁を作る (ステップ 1) . . . . .	35
6.3	パドルを動かす (ステップ 2) . . . . .	35
6.4	ボールを動かす (ステップ 3) . . . . .	36
6.5	ゲームの勝敗を判定する (ステップ 4) . . . . .	36

### Activity 7 シューティングゲーム 39

7.1	作成するゲーム . . . . .	39
7.2	主役を作る (ステップ 1) . . . . .	40
7.3	弾を発射する (ステップ 2) . . . . .	40
7.4	敵たちを作る (ステップ 3) . . . . .	41
7.5	敵たちの移動 (ステップ 4) . . . . .	42
7.6	衝突の定義 (ステップ 5) . . . . .	43
7.7	終了判定 (ステップ 6) . . . . .	43

## Part IV 音楽を演奏しよう 47

### Activity 8 音楽の演奏 48

8.1	メロディの演奏 . . . . .	48
8.2	メロディの記述 . . . . .	48
8.3	楽器の指定 . . . . .	49



8.4	音楽の構造をプログラムする . . . . .	50
8.5	メロディの合奏 . . . . .	52
8.6	楽器を変えて演奏する . . . . .	53
8.7	楽譜からの入力 . . . . .	53
8.8	リズム . . . . .	56
<b>Activity 9</b>	<b>音楽で楽しもう</b>	<b>57</b>
9.1	琉球音階の自動作曲 . . . . .	57
9.2	ランダムな音階の自動作曲 . . . . .	59
9.3	フレーズを利用した自動作曲 . . . . .	59
9.4	リズムと組み合わせた自動作曲 . . . . .	60
<b>Part V</b>	<b>ジェスチャーで操作しよう</b>	<b>63</b>
<b>Activity 10</b>	<b>LeapMotion を使ってみよう</b>	<b>64</b>
10.1	ドリトルと通信するための設定 . . . . .	64
10.2	手の位置と指の情報の取得 . . . . .	65
10.3	ジェスチャーの取得 . . . . .	66
<b>Part VI</b>	<b>センサや LED と入出力しよう</b>	<b>67</b>
<b>Activity 11</b>	<b>Arduino と通信しよう</b>	<b>68</b>
11.1	Arduino の入手 . . . . .	68
11.2	ドリトルと通信するための設定 . . . . .	68
11.3	デジタル出力による動作の確認 . . . . .	69
11.4	アナログ出力 . . . . .	71
11.5	デジタル入力 . . . . .	71
11.6	アナログ入力 . . . . .	72
11.7	Arduino の配線例 . . . . .	73
11.8	加速度センサの応用例 . . . . .	74
<b>Part VII</b>	<b>ロボティストを動かそう</b>	<b>77</b>
<b>Activity 12</b>	<b>Studuino と通信しよう</b>	<b>78</b>
12.1	Studuino の入手 . . . . .	78
12.2	ドリトルと通信するための設定 . . . . .	79

12.3	ドリトルからプログラムを転送する . . . . .	81
12.4	センサーなどのパーツを接続する . . . . .	81
12.5	デジタル出力による動作の確認 . . . . .	83
12.6	アナログ出力 . . . . .	85
12.7	デジタル入力 . . . . .	86
12.8	アナログ入力 . . . . .	87
12.9	ライントレースの例 . . . . .	88
12.10	サーボモーター . . . . .	89

## **Part VIII MYU ロボを動かそう 91**

### **Activity 13 ロボットの準備 92**

13.1	ドリトルとの接続 . . . . .	92
13.2	ロボット . . . . .	92
13.3	対話的な操作による動作確認 . . . . .	93

### **Activity 14 ロボットを動かそう 95**

14.1	プログラムの記述 . . . . .	95
14.2	プログラムの実行 . . . . .	96
14.3	モーターの性質 . . . . .	97
14.4	繰り返し . . . . .	97

### **Activity 15 迷路を抜けよう 99**

15.1	想定するコース . . . . .	99
15.2	移動距離で迷路を抜ける . . . . .	99
15.3	スイッチ入力の検出 . . . . .	100
15.4	ユーザー定義命令 . . . . .	103

### **Activity 16 ものを運ぼう 105**

16.1	想定するコート . . . . .	105
16.2	ロボットの拡張 . . . . .	105
16.3	ものを運ぶプログラム . . . . .	106
16.4	命令の定義 . . . . .	107

## **Part IX ネットワークで通信しよう 111**

### **Activity 17 ネットワーク通信 112**

17.1	サーバーの起動 . . . . .	112
17.2	IP アドレスの確認 . . . . .	112
17.3	サーバーとの接続 . . . . .	113
17.4	オブジェクトの書き込み . . . . .	113
17.5	オブジェクトの読み出し . . . . .	113
<b>Activity 18</b>	<b>チャットを作ろう</b>	<b>115</b>
18.1	メッセージを送信する . . . . .	115
18.2	メッセージを受信する . . . . .	116
18.3	発言に名前を入れる . . . . .	117
18.4	受信の自動化 (1) . . . . .	117
18.5	受信の自動化 (2) . . . . .	118
18.6	チャットプログラム . . . . .	119
<b>Activity 19</b>	<b>音楽を交換しよう</b>	<b>120</b>
19.1	音楽を演奏する . . . . .	120
19.2	音楽を送信する . . . . .	120
19.3	音楽を受信できるようにする . . . . .	121
19.4	長い曲を交換 (ダウンロードとストリーミング) . . . . .	122
<b>Activity 20</b>	<b>ネットワークゲームを作ろう</b>	<b>125</b>
20.1	壁を作る (ステップ 1) . . . . .	125
20.2	パドルを動かす (ステップ 2) . . . . .	126
20.3	ボールの動きを定義する (ステップ 3) . . . . .	126
20.4	通信を準備する (ステップ 4) . . . . .	127
20.5	通信しながらボールを動かす (ステップ 5) . . . . .	127
20.6	ゲームの勝敗を判定する (ステップ 6) . . . . .	129
20.7	プログラムを実行する (ステップ 7) . . . . .	131
<b>Part X</b>	<b>付録</b>	<b>133</b>
<b>付録 A</b>	<b>授業での利用</b>	<b>134</b>
A.1	プログラミング . . . . .	134
A.2	ネットワークプログラミング . . . . .	135
A.3	ロボット制御 . . . . .	136
<b>付録 B</b>	<b>よいプログラムを書くために</b>	<b>138</b>

B.1	よいプログラムって何だろう . . . . .	138
B.2	よいプログラムの作り方 . . . . .	138
<b>付録 C</b>	<b>ドリトル言語の基礎知識</b>	<b>142</b>
C.1	プログラミングとは . . . . .	142
C.2	オブジェクト、プロパティ、変数 . . . . .	143
C.3	メッセージ送信 . . . . .	145
C.4	中置記法 . . . . .	147
C.5	ブロックとメソッド . . . . .	148
C.6	ブロックと制御構造 . . . . .	149
C.7	タイマーとスレッド . . . . .	153
C.8	配列と複数オブジェクトの利用 . . . . .	157
C.9	オブジェクトの親子関係 . . . . .	159
C.10	変数とその束縛 . . . . .	161
C.11	字句の約束 . . . . .	163
<b>付録 D</b>	<b>標準オブジェクト</b>	<b>167</b>
D.1	数値オブジェクト . . . . .	167
D.2	多倍長整数オブジェクト . . . . .	168
D.3	文字列オブジェクト . . . . .	169
D.4	真偽値 . . . . .	170
D.5	色オブジェクト . . . . .	171
D.6	タートルオブジェクト . . . . .	171
D.7	図形オブジェクト . . . . .	172
D.8	GUI 部品 . . . . .	173
D.9	オブジェクトの保存と読み出し . . . . .	175
D.10	テキストファイルの操作 . . . . .	177
<b>付録 E</b>	<b>ドリトルの命令一覧</b>	<b>178</b>
E.1	基本オブジェクト . . . . .	178
E.2	ネットワークオブジェクト . . . . .	198
E.3	グラフィック関係のオブジェクト . . . . .	199
E.4	GUI オブジェクト . . . . .	209
E.5	ショートカットキー一覧 . . . . .	217
E.6	音楽オブジェクト . . . . .	218
E.7	Arduino オブジェクト . . . . .	227
E.8	LeapMotion オブジェクト . . . . .	230





Part I

# ドリトルを使ってみよう



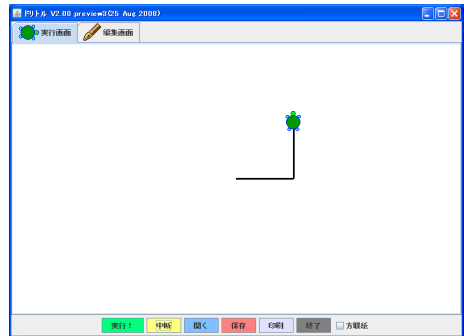
## Activity 1



# はじめてのプログラミング

ドリトルのプログラムを作ってみよう。プログラムは**編集画面**に入力し、「実行！」と書かれた**実行ボタン**を押すことで実行できる。<sup>\*1</sup>

実行！



## 1.1 オブジェクトを作る

ドリトルでは、**オブジェクト**を作り、それに**命令**を送る形でプログラムを動作させる。ここでは**タートルオブジェクト**を使い、**タートルグラフィックス**を扱う。タートルオブジェクトは標準ではカメの姿をしており、歩いた軌跡が線として残る。そこで、画面上を動かすことで絵を描くことができる。

タートルオブジェクトを作るには、**タートル**という名前のオブジェクトに**作る**という命令を送る。文の最後には「。」を書く。

<sup>\*1</sup> プログラムは Ctrl-G でも実行できる。編集画面と実行画面は、Ctrl-T で切り替えられる。Ctrl-O でファイルを開く。文字の大きさは Ctrl-↑、Ctrl-↓、Ctrl-0 で変更できる。Macintosh では Ctrl の代わりに Command キーを使用する。



タートル！ 作る。



実行すると、画面にタートルオブジェクトが表示される。ドリトルではオブジェクトを複製する形で新しいオブジェクトを生成する。そのために、**プロトタイプオブジェクト**と呼ばれるひな形となるためのオブジェクトが用意されている。プロトタイプオブジェクトは画面上には表示されないが、生成したすべてのオブジェクトの親の役割をする重要なオブジェクトである。

この例では、プロトタイプオブジェクトである「タートル」に「作る」を送ることで新しいタートルオブジェクトを生成した。

## 1.2 オブジェクトに名前を付ける

ドリトルでは、オブジェクトを指定して命令を送る。先ほど作ったプログラムでは、生成したオブジェクトに**名前**を付けていないため、画面に表示されたオブジェクトに命令を送ることはできない。オブジェクトに名前を付けるには、

名前 = オブジェクト。

のように、**=** の左辺に名前を書き、右辺にオブジェクトを指定する。

カメ太=タートル！ 作る。



実行すると、先ほどの例と同じく画面にタートルオブジェクトが表示される。見かけは変わらないが、このタートルオブジェクトには「カメ太」という名前が付いている。

名前は**変数**と呼ばれることがある。名前には英字と数字のほか、仮名や漢字などの日本語文字を使うことができる。ただし、名前の先頭に数字を使うことはできず、名前の中に空白や記号を含めることはできない。また、**タートル** といった最初から用意されている名前を使わないように注意する必要がある。

名前が示すオブジェクトは常に1個である。複数のオブジェクトに同じ名前を付けた場合には、最後のオブジェクトだけに名前が残り、他のオブジェクトは名前がなくなってしまう。

### 1.3 オブジェクトに命令する

ドリトルでは、オブジェクトに命令を送ることでプログラムを実行する。命令を送るオブジェクトは！ で指定し、その右側に送る命令を書く。たとえば、

カメ太！ 100 歩く。

は、「カメ太」に対して「100 歩く」という命令を送っている。ここで「歩く」は命令である。「100」は命令とともに送られる値であり、**パラメータ**と呼ばれる。数値に「100 歩」のように単位を付けると、単位は無視されて数字の部分だけが使われる。パラメータと命令の間は**空白**で区切る必要がある。

オブジェクトが理解できる命令はオブジェクトの種類ごとに決まっている。オブジェクトの種類と命令は付録 E にまとめられている。表 1.1 はタートルが理解できる命令の一部である。命令は**メソッド**とも呼ばれる。自分で命令を追加することもできる。この方法は、後の章で扱う。

表 1.1 タートルの命令（一部）

命令	用途	使用例
作る	オブジェクトを作る	カメ太=タートル！ 作る。
歩く	前進する	カメ太！ 100 歩く。
右回り	右に回る	カメ太！ 90 右回り。
左回り	左に回る	カメ太！ 90 左回り。

### 1.4 命令を順に実行する（1）

プログラムは上から順に実行されていく。そこで、複数の命令を順に実行したいときは、上から 1 行ずつ書いていけばよい。実際に書いてみよう。

まず、1 行書いて実行する。画面にはカメが表示される。

カメ太=タートル！ 作る。



1 行追加して実行する。画面のカメが前に歩く。

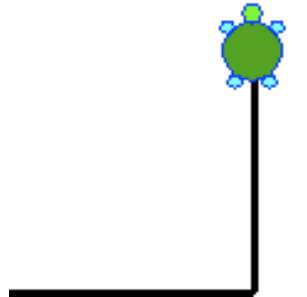
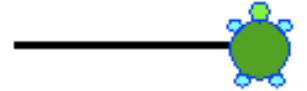
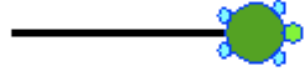
```
カメ太=タートル！ 作る。
カメ太！ 100 歩く。
```

もう 1 行追加して実行する。カメは歩いた後に左を向く。

```
カメ太=タートル！ 作る。
カメ太！ 100 歩く。
カメ太！ 90 左回り。
```

もう 1 行追加して実行する。カメは歩いてから左を向いてさらに歩く。

```
カメ太=タートル！ 作る。
カメ太！ 100 歩く。
カメ太！ 90 左回り。
カメ太！ 100 歩く。
```



## 1.5 命令を順に実行する (2)

オブジェクトは命令を実行すると、結果としてオブジェクトを返す。たとえば、

```
カメ太=タートル！ 作る。
```

というプログラムは、「タートル！ 作る」という命令を実行した結果、新しいタートルオブジェクトが返される。

この文では、生成された新しいオブジェクトに「カメ太」という名前を付けていた。このように、新しいオブジェクトを生成したり数式を計算したりする場合は、返された結果の値に名前を付けたり画面に表示したりして活用することができる。

一方、タートルを動かすような命令では、元のオブジェクトがそのまま返されることが多い。そこで、

```
カメ太！ 100 歩く 90 右回り。
```

のように命令を並べて書くことで、ひとつのオブジェクトに続けて複数の命令を送り、実行することができる。このように、命令を実行した結果のオブジェクトに続けて命令を送ることを**カス**

## 6 Activity 1 はじめてのプログラミング

**ケード**という。ここで、「90 右回り」は、「カメ太！ 100 歩く」から返されたオブジェクトに送られている。

たとえば、

タートル！ 作る 100 歩く。

では、「タートル！ 作る」から返されたオブジェクトに対して「100 歩く」が送られる。つまり、新しく作られたオブジェクトが 100 歩歩くことになる。

次のプログラムは、1 行に 1 命令ずつ書いたプログラムである。内容の割にすぐに行数が長くなってしまい無駄が多い。

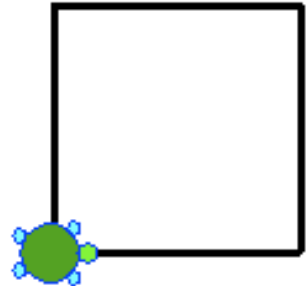
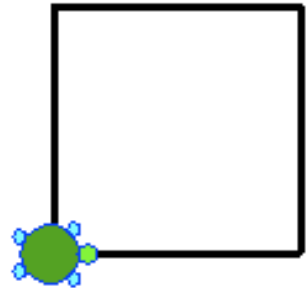
```
カメ太=タートル！ 作る。  
カメ太！ 100 歩く。  
カメ太！ 90 左回り。  
カメ太！ 100 歩く。  
カメ太！ 90 左回り。  
カメ太！ 100 歩く。  
カメ太！ 90 左回り。  
カメ太！ 100 歩く。  
カメ太！ 90 左回り。
```

次のプログラムは、できるだけ続けて書いたプログラムである。全体で 2 行と短くなったが、横に長くなり、分かりづらくなってしまった。

```
カメ太=タートル！ 作る。  
カメ太！ 100 歩く 90 左回り 100 歩く 90 左回り  
100 歩く 90 左回り 100 歩く 90 左回り。
```

次のプログラムは、動作ごとにまとめて書いたプログラムである。このように、ひとまとまりの動作を 1 行にまとめて書くと分かりやすくなる。

```
カメ太=タートル！ 作る。  
カメ太！ 100 歩く 90 左回り。  
カメ太！ 100 歩く 90 左回り。  
カメ太！ 100 歩く 90 左回り。  
カメ太！ 100 歩く 90 左回り。
```



## 1.6 繰り返し

コンピュータは命令を高速に実行し、大量の命令を実行しても繰り返すことに疲れたり飽きたりすることはない。そこで、ある処理を何度も繰り返すプログラムはよく使われている。先ほどのプログラムにも、同じ記述が何度も出てきていた。

ドリトルでは、プログラムの一部を「「...」」で囲むことで、**ブロックオブジェクト**として扱うことができる。たとえば、

```
「カメ太！ 100 歩く 90 左回り」
```

はブロックオブジェクトである。ブロックの中には複数の文を書くことができる。ブロックの末尾では、文末の「。」を省略してもよい。

ブロックに対して自分自身を何度も実行させると**繰り返し**を実現できる。一定回数の繰り返しは「「...」！ 3回 繰り返す。」のように記述する。ここで、数字に続く数字以外の文字は無視されるため、「3回」は「3」と同じ意味である。

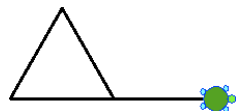
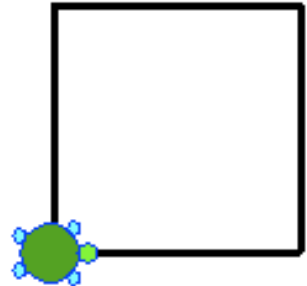
次のプログラムは、繰り返しを使った例である。

```
カメ太＝タートル！ 作る。  
「カメ太！ 100 歩く 90 左回り」！ 4 繰り返す。
```

この例では、繰り返しを使うことで、四角を描く4行のプログラムを1行にまとめることができた。このように、同じ処理が何度も書かれているときは、繰り返しを使うことで、プログラムを簡潔にまとめることができる。

繰り返しの結果として、最後に実行された文の値が返される。よって、次のような記述も可能である。このプログラムを実行すると、三角形を描いた後で、カメ太が200歩歩く。

```
カメ太＝タートル！ 作る。  
「カメ太！ 100 歩く 120 左回り」！ 3回 繰り返す  
200 歩く。
```



## 1.7 命令の定義

タートルなどのオブジェクトは、最初から「歩く」、「右回り」などの命令を使えるが、オブジェクトに新しい**命令（メソッド）**を追加して使うこともできる。

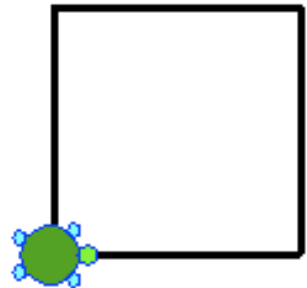
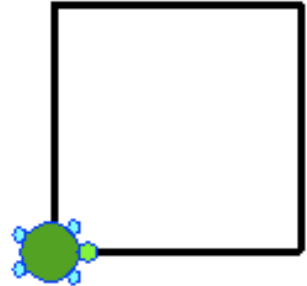
次のプログラムは正方形を描くプログラムである。よく読めば「100 歩歩いて 90 度左に回る動作を 4 回繰り返すから正方形が描かれるのだな」と分かるが、慣れないと分かりづらい。

カメ太=タートル！ 作る。  
「カメ太！ 100 歩く 90 左回り」！ 4 繰り返す。

「正方形」という命令をカメ太が理解できるように定義してみよう。オブジェクトに新しい命令を追加するには、追加したい命令を「「...」」で囲んだ**ブロック**で定義する\*2。次のプログラムでは、カメ太に正方形という名前の命令を定義した。命令定義のブロックでは、オブジェクト自身（ここではカメ太）を**自分**と書くことと便利である。定義した命令は、普通の命令と同様に使うことができる。

カメ太=タートル！ 作る。  
カメ太：正方形=「「自分！ 100 歩く 90 左回り」！ 4 繰り返す」。  
カメ太！ 正方形。

作った命令は、プログラムの中で何度も使うことができる。次のプログラムでは、正方形を 3 回描いている。命令を定義して使うことで、プログラムを見たときに「ここで正方形を描いている」ということが一目瞭然であり、プログラムも短く書くことができた。



\*2 詳しくは付録 C のブロックの説明を参照。

カメ太=タートル！ 作る。

カメ太：正方形=「「自分！ 100 歩く 90 左回り」！ 4  
繰り返す」。

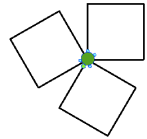
カメ太！ 正方形。

カメ太！ 120 左回り。

カメ太！ 正方形。

カメ太！ 120 左回り。

カメ太！ 正方形。



追加した命令には、「100 歩く」と同じようにパラメータを渡すことができる。パラメータを受け取るためには、変数をブロックの先頭で「|...|」という記号の間に記述する。複数のパラメータを受け取る時は空白で区切る。次のプログラムでは、パラメータとして辺の長さを受け取り、1 辺がその長さの正方形を描いている。

カメ太=タートル！ 作る。

カメ太：正方形=「|x|「自分！ (x) 歩く 90 左回り」！ 4  
繰り返す」。

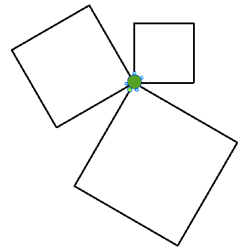
カメ太！ 100 正方形。

カメ太！ 120 左回り。

カメ太！ 150 正方形。

カメ太！ 120 左回り。

カメ太！ 200 正方形。



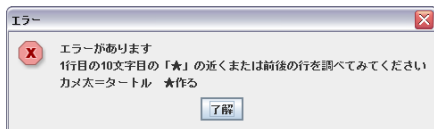
## ●プログラムのデバッグ

エラーを直すことを**デバッグ**と呼ぶ。

**文法エラー**は、文法に正しくない部分があるために、ドリトルがプログラムを解釈できないエラーである。実行ボタンを押すと、実行が行われる前にエラーがダイアログで表示される。ダイアログには、コンピュータがエラーに気づいた大まかな位置が示される。

次の例では、「作る」の前に「!」を書き忘れていることがエラーとして検出されている。

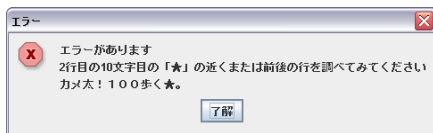
カメ太=タートル 作る。



文法エラーが表示された場合は、表示された場所の近くを確認する。プログラムの行は、編集画面の左に表示されている。また、編集画面の上部にはカーソル位置が「行：列」の形で表示されている。

ドリトルを学んでいる際のエラーとして最も多いのは、命令の前などに**空白**を入れ忘れるエラーと、文末の「。」を入れ忘れるエラーである。次の例は、命令の前に空白を入れ忘れたために、「100 歩く」という部分でエラーになっている。

カメ太！ 100 歩く。



次の例は、1行目の末尾に「。」を書き忘れた場合である。「。」を書き忘れた場合には、次の文とつながってしまい、コンピュータが矛盾を発見した場所でエラーが表示される。そのため、必ずしも「。」があるべき場所がエラーにならないことに注意が必要である。

カメ太=タートル 作る  
カメ太！ 100 歩く。

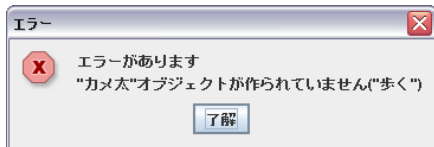


**実行時エラー**は、プログラムの実行中に起きるエラーである。文法的に正しくても「命令のパラメータが正しくない」「オブジェクトが解釈できない命令が送られた」などの原因で、実行中にエラーになることがある。実行時エラーでは、エラーの起きた命令名とオブジェクト名がダイアログに表示される。それを手がかりに、エラーの箇所を探す必要がある。

まだ作られていないオブジェクトに命令が送られたり、オブジェクトの名前を間違えた場合には、正しいオブジェクトに命令を送ることができずに実行時エラーが発生する。次の例は、「カメ太」を作らずに「歩け」という命令を送った場合のメッセージである。

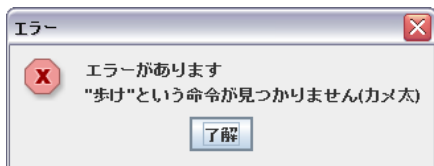
カメ太！ 100 歩く。





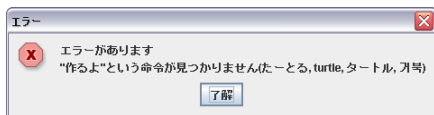
オブジェクトに理解できない命令が送られた場合にも、実行時エラーが発生する。次の例では、「歩く」を「歩け」と書き間違えたため、タートルオブジェクトであるカメ太は命令を理解できない。

カメ太=タートル！ 作る。  
カメ太！ 100 歩け。



同様のエラーは、色オブジェクトを括弧で囲まずに指定した場合や、図形オブジェクト用の「塗る」をタートルに送った場合などにも発生する。

なお、オブジェクトは複数の名前で参照されることがあるため、エラーダイアログでは複数の名前が表示されることがある。次の画面では、タートルオブジェクトの名前がカタカナとひらがなを含む複数の言語で表示されている。



プログラムの書き方については「よいプログラムを書くために」(付録 B) も参照されたい。



Part II

# 絵を描こう



## Activity 2



# 描いた絵に色を塗ろう

Activity 1 では、タートルオブジェクトを動かして、画面に図形を描いたり、図形を描くための命令を定義した。この Activity では、描いた図形をオブジェクトとして操作する方法を扱う。

## 2.1 図形の生成

タートルが移動して描いた線は、タートルの一部である。カメラのしっぽが伸びている状態をイメージすると分かりやすい。

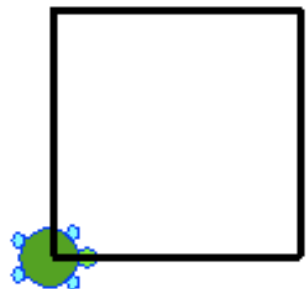
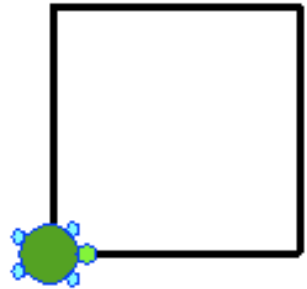
カメラ＝タートル！ 作る。  
「カメラ！ 100 歩く 90 左回り」！ 4 繰り返す。

**図形を作る**という命令を使うと、しっぽをカメラから切り離して、新たに**図形**オブジェクトを作れる。自分で描いた絵をオブジェクトにすることで、アニメーションやゲームなどで動かして活用できる。

次の例では、描いた図形をオブジェクトにして、「四角」という名前を付けた。

カメラ＝タートル！ 作る。  
「カメラ！ 100 歩く 90 左回り」！ 4 繰り返す。  
四角＝カメラ！ 図形を作る。

「図形を作る」のパラメータとして色を指定すると、その色で塗られた図形が作られる。次の例では青く塗られた図形を作った後、**位置**で画面上の特定の座標（ここでは (100,100)）に移動している。色については 2.3 節で説明する。数式や変数は括弧 (...)



で囲んで記述する。

カメ太=タートル！ 作る。  
「カメ太！ 100 歩く 90 左回り」！ 4 繰り返す。  
四角=カメ太！（青）図形を作る。  
四角！ 100 100 位置。



ドリトルの画面上の位置は  $xy$  座標で指定できる。原点は画面の中央で、右に  $x$  軸、上に  $y$  軸が伸びている。数学で扱う座標と同様である。

図形オブジェクトでは、図形を描き始めた点が図形の座標になる。タートルオブジェクトでは画像の中心が、Activity 3 で扱うボタンなどの **GUI 部品** は左上がオブジェクトの座標である。

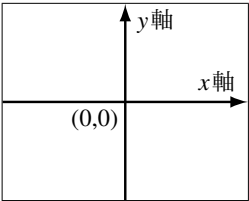


表 2.1 は図形の命令の一部である。

表 2.1 図形の命令（一部）

命令	用途	使用例
右回り	右に回る	四角！ 30 右回り。
左回り	左に回る	四角！ 30 左回り。
移動する	移動する	四角！ 100 0 移動する。
位置	特定の位置に動く	四角！ 100 100 位置。
作る	オブジェクトを複製する	四角 2 = 四角！ 作る。
塗る	色を塗る	四角！（青）塗る。

## 2.2 図形の複製

オブジェクトは複製して使うことができる。あるオブジェクトを複製すると、同じオブジェクトが作られる。

複製を作る命令は**作る**である。通常のプログラムでは「カメ太=タートル！ 作る」のように、新しいオブジェクトを作るために使われるが、「カメ太！ 作る」とすると、画面上のカメ太が複製されて、もうひとつのタートルオブジェクトが作られる。

カメ太=タートル！ 作る。  
カメ吉=カメ太！ 作る。

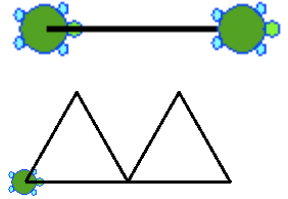


タートルのように画面に表示されているオブジェクトでは、複製されたオブジェクトは元のオブジェクトと同じ位置に作られるため、ぴったり重なり合ってしまう、見た目には複製されたことが分かりづらいことがある。このようなときは、どちらか片方を動かしてみればよい。次のプログラムでは、新しく作られたカメラ吉を 100 歩動かしている。

```
カメラ太=タートル！ 作る。
カメラ吉=カメラ太！ 作る 100 歩く。
```

図形も同様に複製できる。自分の好きな形を描いたら、複製して画面の上で増やしてみよう。次のプログラムは、三角形を複製して横に並べている。

```
カメラ太=タートル！ 作る。
三角=「カメラ太！ 100 歩く 120 左回り」！ 3 繰り返す 図形を作る。
三角！ 作る 100 0 移動する。
```



## 2.3 色オブジェクト

図形を塗るときの色もオブジェクトである。基本的な色として、8色（黒、赤、緑、青、紫、水色、黄色、白）の色オブジェクトがあらかじめ用意されている。次のプログラムは、三角形の図形オブジェクトを「緑」に塗るプログラムである。

```
カメラ太=タートル！ 作る。
「カメラ太！ 100 歩く 120 右回り」！ 3 繰り返す。
三角=カメラ太！（緑）図形を作る。
```

あらかじめ用意されている色は、もっとも明るい状態である。必要に応じて**暗くする**で暗くして使うことができる。暗くした色は**明るくする**で明るくすることができる。次のプログラムは、「暗くする」で「濃い緑」を作り、色を塗っている。



カメ太=タートル！ 作る。  
「カメ太！ 100 歩く 120 右回り」！ 3 繰り返す。  
濃い緑=緑！ 暗くする。  
三角=カメ太！（濃い緑）図形を作る。

色を混ぜ合わせて新しい色を作ることができる。次のプログラムは、「光」パレットを使い、「緑」と「黄色」を混ぜて「きみどり」という新しい色を作っている\*1。

カメ太=タートル！ 作る。  
「カメ太！ 100 歩く 120 右回り」！ 3 繰り返す。  
きみどり=光！（緑）（黄色）混ぜる。  
三角=カメ太！（きみどり）図形を作る。

さらに黄色に近い色にしたければ、次のように、同じ色を 2 回以上加えたり、

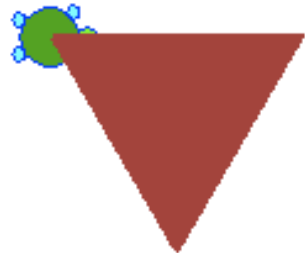
きみどり=光！（緑）（黄色）（黄色）混ぜる。

次のように、3 個以上の色を加えることもできる。

きみどり=光！（緑）（黄色）（青）混ぜる。

特殊な色を作りたい場合は、「赤」、「緑」、「青」の**三原色**を数値で指定して新しい色オブジェクトを作る。色の強さは 0~255 で表現する。0 は光がまったくない暗い状態、255 はいちばん明るい状態である\*2。

カメ太=タートル！ 作る。  
「カメ太！ 100 歩く 120 右回り」！ 3 繰り返す。  
茶色=色！ 166 42 42 作る。  
三角=カメ太！（茶色）図形を作る。



\*1 いろいろな色の光を混ぜ合わせていくと、明るくなってやがて白に近づく。このような混ぜ合わせは**加法混色**である。絵具を混ぜ合わると黒に近づく。このような混ぜ合わせは**減法混色**である。詳しくは付録 E を参照。

\*2 色の強さは 16 進数でも指定できる。詳しくは付録 E を参照。

## Activity 3



# ペイントソフトを作ろう

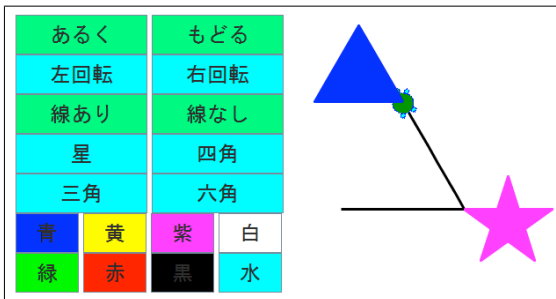
この Activity では、ボタンで絵を描くペイントソフトを作る。

### 3.1 作成するペイントソフト

次の図は製作例である。画面にはタートルが表示されており、左側にはボタンが並んでいる。ボタンを押すと、次のような操作を行える。

- タートルの操作。この例では、「あるく」、「もどる」、「左回転」、「右回転」、「線あり」、「線なし」が用意されている。
- 描いた線で図形を作る。この例では「図形を作る」が用意されている。
- 図形の描画。この例では、「星」、「四角」、「三角」、「六角」が用意されている。
- 色。この例では、基本的な 8 色が用意されており、ボタン自体がそれぞれの色で塗られている。

このようなペイントソフトを、少しずつ作っていこう。





## 3.2 ボタンの生成

今まで作ってきたプログラムは、いちど実行ボタンを押して実行したら、プログラムが動くのを黙って見ているしかなかった。また、コンピュータは高速に動くので、描いている途中は見えずに、一瞬で結果の状態が表示されてしまっていた。

コンピュータの画面で動く多くのアプリケーションソフトのように、ボタンなどの **GUI 部品**（プログラムの画面に置いて対話的に使うグラフィカルな部品）を使えると便利である。そこでここでは、**ボタン**オブジェクトを使い、マウスでプログラムの動きを制御する方法を学ぶ。

タートルなどのオブジェクトと同様に、画面にボタンオブジェクトを生成することができる。ボタンに表示する文字は、**作る**のパラメータとして与える。次のプログラムは、「カメ太」というタートルオブジェクトと、「前進ボタン」というボタンオブジェクトを画面に作っている。

```
カメ太=タートル！ 作る。
前進ボタン=ボタン！ "前進" 作る。
```

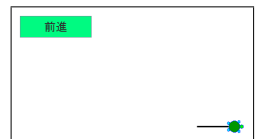
実行すると、無事に画面にボタンが表示された。しかし、押しても何も起きないだろう。これは、押したときに何をしたらよいかという動作をボタンに指示していないためである。



## 3.3 動作の定義

ボタンオブジェクトに**動作**という名前の命令（メソッド）を定義することで、ボタンを押したときの動作を定義できる。次のプログラムは、「前進ボタン」に「動作」を定義している。実行すると、ボタンを押すたびにカメ太が 100 歩ずつ前進する。

```
カメ太=タートル！ 作る。
前進ボタン=ボタン！ "前進" 作る。
前進ボタン：動作=「カメ太！ 100 歩く」。
```



以上が、ボタンの基本的な使い方である。表 3.1 に、ボタンの命令の一部をまとめておく。

表 3.1 ボタンの命令（一部）

命令	用途	使用例
移動する	移動する	前進ボタン！ 100 0 移動する。
位置	特定の位置に動く	前進ボタン！ 100 100 位置。
大きさ	ボタンの大きさを指定する	前進ボタン！ 150 100 大きさ。
塗る	ボタンの色を指定する	前進ボタン！（緑）塗る。
文字色	文字の色を指定する	前進ボタン！（青）文字色。

3.4 タートルを操作する

ここからは、ペイントソフトを作っていく。最初に、主役となるタートルを作ろう。ここでは「カメ太」という名前にした。次に、画面の左上に 2 つのボタンを作る。「あるく」ボタンを押すと、カメ太は前進する。「もどる」ボタンを押すと、カメ太は後退する。先頭の行は、コメントである。「//」からその行の最後まで人は読むためのコメントとして扱われ、プログラムとして解釈されない。

```
// タートルオブジェクトを操作する。
カメ太=タートル！ 作る。
歩くボタン=ボタン！ "あるく" 作る。
歩くボタン：動作＝「カメ太！ 20 歩く」。
戻るボタン=ボタン！ "もどる" 作る。
戻るボタン：動作＝「カメ太！ 20 戻る」。
```



続いて、タートルを回転するボタンを作る。「左回転」ボタンを押すと、カメ太は左に回転する。「右回転」ボタンを押すと、カメ太は右に回転する。



左回転ボタン=ボタン！ "左回転" 作る（水）塗る 次の行。

左回転ボタン：動作＝「カメ太！ 30 左回り」。

右回転ボタン=ボタン！ "右回転" 作る（水）塗る。

右回転ボタン：動作＝「カメ太！ 30 右回り」。

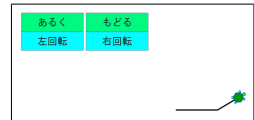
続いて、タートルが動いたときに線を描くかどうかを指定するボタンを作る。「線あり」 ボタンを押すと、カメ太は動いたときに線を描くようになる。「線なし」 ボタンを押すと、カメ太は動いたときに線を描かないようになる。

線ありボタン=ボタン！ "線あり" 作る 次の行。

線ありボタン：動作＝「カメ太！ ペンあり」。

線なしボタン=ボタン！ "線なし" 作る。

線なしボタン：動作＝「カメ太！ ペンなし」。



### 3.5 図形を描く

続いて、図形を描くボタンを作る。「星」 ボタンを押すと、カメ太が星を描き、図形オブジェクトにする。同様に、「四角」、「三角」、「六角」 ボタンを用意して、三角形、四角形、六角形の図形オブジェクトをボタンひとつで作れるようにする。



// 図形を描き図形オブジェクトにする

星ボタン=ボタン！ "星" 作る（水）塗る 次の行。

星ボタン：動作＝「カメ太！ 100 歩く 144 右回り」！ 5 繰り返す」。

四角ボタン=ボタン！ "四角" 作る（水）塗る。

四角ボタン：動作＝「カメ太！ 100 歩く 90 左回り」！ 4 繰り返す」。

三角ボタン=ボタン！ "三角" 作る（水）塗る 次の行。

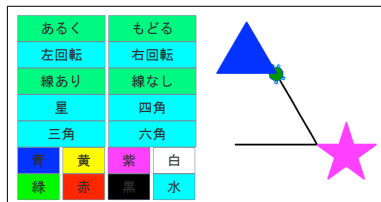
三角ボタン：動作＝「カメ太！ 100 歩く 120 左回り」！ 3 繰り返す」。

六角ボタン=ボタン！ "六角" 作る（水）塗る。

六角ボタン：動作＝「カメ太！ 100 歩く 60 左回り」！ 6 繰り返す」。

### 3.6 図形に色を塗る

続いて、図形に色を塗るボタンを作る。ボタンは「青、黄色、紫、白、緑、赤、黒、水色」の8色分を作り、4個ずつ2列に並べている。それぞれの色のボタンを押すと、描いた図形に色が塗られる。



// 図形オブジェクトに色を塗る

青ボタン=ボタン！ "青" 作る 72 45 大きさ (青) 塗る 次の行。

青ボタン：動作＝「カメ太！（青）図形を作る」。

黄ボタン=ボタン！ "黄" 作る 72 45 大きさ (黄) 塗る。

黄ボタン：動作＝「カメ太！（黄）図形を作る」。

紫ボタン=ボタン！ "紫" 作る 72 45 大きさ (紫) 塗る。

紫ボタン：動作＝「カメ太！（紫）図形を作る」。

白ボタン=ボタン！ "白" 作る 72 45 大きさ (白) 塗る。

白ボタン：動作＝「カメ太！（白）図形を作る」。

緑ボタン=ボタン！ "緑" 作る 72 45 大きさ (緑) 塗る 次の行。

緑ボタン：動作＝「カメ太！（緑）図形を作る」。

赤ボタン=ボタン！ "赤" 作る 72 45 大きさ (赤) 塗る。

赤ボタン：動作＝「カメ太！（赤）図形を作る」。

黒ボタン=ボタン！ "黒" 作る 72 45 大きさ (黒) 塗る。

黒ボタン：動作＝「カメ太！（黒）図形を作る」。

水ボタン=ボタン！ "水" 作る 72 45 大きさ (水) 塗る。

水ボタン：動作＝「カメ太！（水）図形を作る」。

## Activity 4



# アニメーション

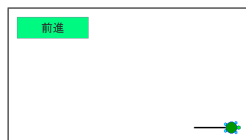
### 4.1 今まで学んだ繰り返し

コンピュータはある動作を繰り返し実行するのが得意である。Activity 1 では、**繰り返し**を使うことで、プログラムの一部を複数回実行できることを学んだ。繰り返されるプログラムは、一瞬で実行される。次のプログラムでは、10 回繰り返している動作は見えず、画面には 10 回繰り返された後の結果が表示される。

カメ太=タートル！ 作る。  
「カメ太！ 20 歩く」！ 10 繰り返す。

Activity 3 では、**ボタン**オブジェクトを使うことで、特定の動作を手動で実行できることを学んだ。次のプログラムでは、画面のボタンを何度も押すことにより、繰り返し実行させることができる。

カメ太=タートル！ 作る。  
前進ボタン=ボタン！ "前進" 作る。  
前進ボタン：動作=「カメ太！ 20 歩く」。



### 4.2 タイマーオブジェクト

**タイマーオブジェクト**を使うと、一定間隔の繰り返しを行うことができる。タイマーは画面に表示されないオブジェクトで、他のオブジェクトに命令を繰り返して伝えるために使われる。

次のプログラムでは、「時計」という名前のタイマーオブジェクトを作り、実行したい内容をブロックで渡して実行している。

このプログラムを実行すると、カメ太は一定の間隔で少しずつ前進する。

```
カメ太=タートル！ 作る。
時計=タイマー！ 作る。
時計！「カメ太！ 20 歩く」実行。
```

作成と実行を1行で書くこともできる。この例では、タイマーに「時計」といった名前を付けずに実行している。

```
カメ太=タートル！ 作る。
タイマー！ 作る「カメ太！ 20 歩く」実行。
```

タイマーには、繰り返す**間隔**と**回数**を指定できる。次のプログラムでは、「時計」という名前のタイマーに間隔と回数を指定している標準では0.1秒間隔で100回の繰り返しを行う。<sup>\*1</sup>

```
カメ太=タートル！ 作る。
時計=タイマー！ 作る 1秒 間隔 5回 回数。
時計！「カメ太！ 20 歩く」実行。
```

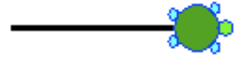
回数は実行時に指定することも可能である。

```
カメ太=タートル！ 作る。
時計=タイマー！ 作る 1秒 間隔。
時計！「カメ太！ 20 歩く」5回 実行。
```

回数の代りに時間を指定することもできる。次のプログラムでは、1秒間隔で5秒間の実行を行う。実行される回数は「5秒間 ÷ 1秒 = 5回」である。

```
カメ太=タートル！ 作る。
時計=タイマー！ 作る 1秒 間隔 5秒 時間。
時計！「カメ太！ 20 歩く」実行。
```

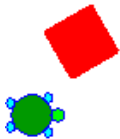
次のプログラムは、ゆっくりと円を描くアニメーションの例である。



<sup>\*1</sup> 「0.1秒 × 100回 = 10秒」であることから、実行に要する時間は10秒間である。

カメ太=タートル！ 作る。  
時計=タイマー！ 作る 0.1秒 間隔 36秒 時間。  
時計！「カメ太！ 1 歩く 1 右回り」実行。

次のプログラムは、図形が回転しながら動くアニメーションの例である。



カメ太=タートル！ 作る。  
四角=「カメ太！ 30 歩く 90 左回り」！ 4 繰り返す (赤) 図形を作る。  
時計=タイマー！ 作る。  
時計！「四角！ 30 右回り 2 2 移動する」実行。

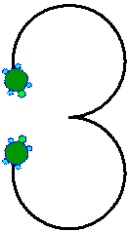
表 4.1 に、タイマーの命令の一部をまとめておく。

表 4.1 タイマーの命令 (一部)

命令	用途	使用例
間隔	実行間隔を指定する。単位は秒	時計！ 0.5 間隔。
回数	実行回数を指定する	時計！ 10 回数。
時間	実行時間を指定する。単位は秒	時計！ 5 時間。
実行	タイマーを実行する	時計！「カメ太！ 10 歩く」実行。
待つ	タイマーの終了を待つ	時計！ 待つ。

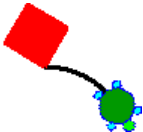
4.3 複数のオブジェクトを動かす

実行するブロックに複数の文を書くことで、複数の動作を同時に行うことができる。次のプログラムは、「カメ太」と「カメ吉」という 2 つのタートルオブジェクトをタイマーの中で同時に動かしている。



カメ太=タートル！ 作る。  
カメ吉=タートル！ 作る。  
時計=タイマー！ 作る 0.1秒 間隔 36秒 時間。  
時計！「カメ太！ 1 歩く 1 右回り。カメ吉！ 1 歩く 1 左回り」実行。

同様に、違う種類のオブジェクトを同時に動かすことも可能である。次のプログラムでは、タートルオブジェクト (カメ太) と図形オブジェクト (四角) を同時に動かしている。



カメ太=タートル！ 作る。

四角=「カメ太！ 30 歩く 90 左回り」！ 4 繰り返す (赤) 図形を作る。

時計=タイマー！ 作る 0.1秒 間隔 36秒 時間。

時計！「カメ太！ 1 歩く 1 右回り。四角！ 1 左回り」実行。

## 4.4 タイマーで複数の実行を行う

タイマーに複数の実行を送ると、それらは順次実行される。次のプログラムではタートルオブジェクト (カメ太) を操作している。最初は右回りに動かし、続いて左回りに動かし、最後に前進させている。実行の前に数値を指定した場合には、その回数だけ実行される。

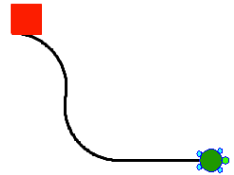
カメ太=タートル！ 作る。

時計=タイマー！ 作る。

時計！「カメ太！ 1 歩く 1 右回り」実行。

時計！「カメ太！ 1 歩く 1 左回り」実行。

時計！「カメ太！ 10 歩く」10 実行。



## 4.5 タイマーの終了を待つ

通常、プログラムは上から順に 1 行ずつ実行され、直前の行の実行が終るのを待って次の行が実行される。一方、タイマーで繰り返されるプログラムは、他のプログラムと並行して動作する。これは、タイマーは通常「数秒間から数分間」という長い時間動作を続けるため、その終了を待っていると、他の動作が行えなくなってしまうためである。このように、ひとつのプログラムの中で同時に複数の処理を並行して実行する仕組みを**スレッド**と呼ぶ。

しかし、ときにはタイマーの実行が終了するのを待ってから次に進みたいことがある。たとえば、ゲームの終了を待って得点を表示するときなどが考えられる。ここで、「タートルをその場で回転させてから、前進する」プログラムを考える。次のプログラムを実行すると、タイマーの実行が終る前にプログラムは次の行に進んでしまい、結果として回転する前にタートルが歩いてし



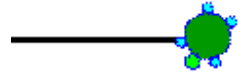
もう。

カメ太=タートル！ 作る。  
時計=タイマー！ 作る。  
時計！「カメ太！ 10 右回り」実行。  
カメ太！ 100 歩く。

実行中のタイマーで**待つ**を実行すると、そのタイマーが終了するまで次の命令に進まない。このように、他のスレッドと実行のタイミングを合わせることを**同期**と呼ぶ。

次のプログラムを実行すると、タイマーの実行が終了するのを待ってから次の行に進むので、タートルは回転してから歩くという正しい動作を行う。

カメ太=タートル！ 作る。  
時計=タイマー！ 作る。  
時計！「カメ太！ 10 右回り」実行。  
時計！ 待つ。  
カメ太！ 100 歩く。





Part III

# ゲームを作ろう



## Activity 5

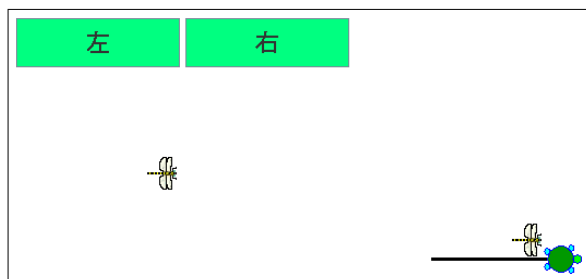


# 宝物拾いゲーム

これまでの Activity では、タートルを移動させることによるグラフィックスと、マウスでクリックしてプログラムを対話的に操作するためのボタン、タイマーによる繰り返し実行などを学んだ。この Activity では、これまでに学んだことを使って、簡単なゲームプログラムを作る。

## 5.1 作成するゲーム

作成するゲームは宝物拾いゲームである。ボタンでタートルを操作して、宝物を拾い集めていく。



画面にはカメとトンボの姿をしたタートルが表示されており、上に 2 個のボタンが並んでいる。タートルはタイマーオブジェクトにより自動的に前進する。トンボは乱数により、実行するたびに画面の異なる位置に表示される。ユーザーは、タートルの向きを左右に変える 2 つのボタンを操作して、タートルを効率よく動かしてトンボのおもちゃを拾う。

## 5.2 タートルを操作する（ステップ 1）

まず、画面にタートルを作る。ここでは「カメ太」という名前にした。次に、画面の上に 2 つのボタンを作る。「左」ボタンを押すと、カメ太はその場で左に 30 度回転する。「右」ボタンを押すと、カメ太はその場で右に 30 度回転する。

ボタンの作成時に、表示するラベルの後にキーを表す文字列（ショートカットキー<sup>\*1</sup>）を指定すると、キーボードからボタンを操作できる。ここでは **LEFT**、**RIGHT** を指定することで、ボタンを押すのと同じ動作を左右の矢印キーで行えるようにした。

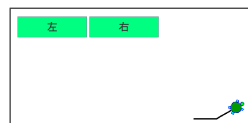
```
// タートルを操作する（ステップ1）
カメ太=タートル！ 作る。
左ボタン=ボタン！ "左" "LEFT" 作る。
左ボタン：動作=「カメ太！ 30 左回り」。
右ボタン=ボタン！ "右" "RIGHT" 作る。
右ボタン：動作=「カメ太！ 30 右回り」。
```



## 5.3 タートルを前進させる（ステップ 2）

次に、タートルを前進させる。次のプログラムでは、「時計」という名前のタイマーオブジェクトを作り、カメ太を 10 歩ずつ前進させる動作を 200 回繰り返す。繰り返す間隔は、標準の 0.1 秒である。よって、20 秒間動作することになる（0.1 秒×200 回）。

```
// タートルを前進させる（ステップ2）
時計=タイマー！ 作る 200 回数。
時計！「カメ太！ 10 歩く」実行。
```



## 5.4 宝物を画面に置く（ステップ 3）

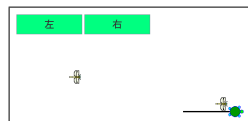
次に、トンボの姿をしたタートルオブジェクトを画面に置く。これらのオブジェクトは、画面に置いた後は操作しないため、特に名前を付ける必要はない。タートルの姿は**変身する**で変更する

<sup>\*1</sup> 詳しくは付録 E を参照。

ことができる。姿は JPEG, GIF, PNG などの形式の画像ファイルで指定することが可能であり、あらかじめいくつかの画像ファイルが用意されている<sup>\*2</sup>。

画面に置く位置は、**乱数**で指定している。横の位置を -299～300 の幅 600 の範囲に、縦の位置を -149～150 の高さ 300 の範囲にするために、それぞれ 1～600 の乱数と 1～300 の乱数を生成し、その値から 300 と 150 を引いて利用している。乱数や引き算などの数式は、括弧 (...) で囲んで記述する。

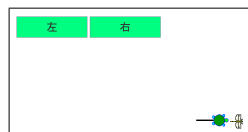
```
// 宝物を画面に置く (ステップ3)
宝=タートル! 作る "tonbo.gif" 変身する ペンなし。
宝! (乱数(600)-300) (乱数(300)-150) 位置。
宝! 作る (乱数(600)-300) (乱数(300)-150) 位置。
宝! 作る (乱数(600)-300) (乱数(300)-150) 位置。
```



## 5.5 宝物を拾う (ステップ 4)

前のプログラムでは、カメ太がトンボに重なっても何も起きず、カメ太はトンボを通り抜けてしまった。ここでは、カメ太がトンボに衝突したときに、トンボの姿を消すことで拾ったことを表現する。次のプログラムでは、カメ太に**衝突メソッド**を定義し、衝突したときに相手に**消える**命令を送っている。

```
// 宝物を拾う (ステップ4)
カメ太:衝突=「|相手| 相手! 消える」。
```



最後にステップ 1 からステップ 4 までの全体のプログラムを掲載しておく。

<sup>\*2</sup> ローカル版では、ドリトルのフォルダに画像ファイルを置くことで、独自の画像ファイルを使うことも可能である。

// タートルを操作する（ステップ1）

カメ太=タートル！ 作る。

左ボタン=ボタン！ "左" "LEFT" 作る。

左ボタン：動作=「カメ太！ 30 左回り」。

右ボタン=ボタン！ "右" "RIGHT" 作る。

右ボタン：動作=「カメ太！ 30 右回り」。

// タートルを前進させる（ステップ2）

時計=タイマー！ 作る 200 回数。

時計！「カメ太！ 10 歩く」実行。

// 宝物を画面に置く（ステップ3）

宝=タートル！ 作る "tonbo.gif" 変身する ペンなし。

宝！（乱数（600）-300）（乱数（300）-150）位置。

宝！ 作る（乱数（600）-300）（乱数（300）-150）位置。

宝！ 作る（乱数（600）-300）（乱数（300）-150）位置。

// 宝物を拾う（ステップ4）

カメ太：衝突=「| 相手 | 相手！ 消える」。

## Activity 6



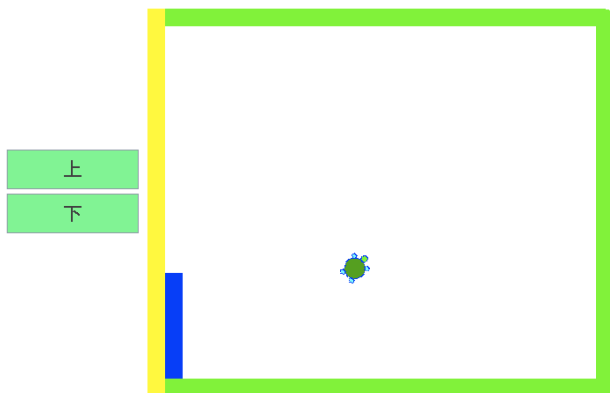
# ピンポンゲーム

この Activity では、ボールをラケットで打ち返すピンポンゲームを作ってみる。

## 6.1 作成するゲーム

画面には四角い枠が描かれており、その中にボールの役割をするタートルと、長方形のラケットがある。ボールはラケットや壁にぶつくと跳ね返る。ラケットは左側の 2 個のボタンで上下に操作する。

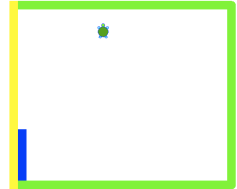
ボールを制限時間の間、ラケットで打ち返し続けられたらクリアとなる。一回でもラケットの後ろの壁にボールがぶつかると、ゲームオーバーとなる。





## 6.2 壁を作る (ステップ 1)

最初に、ゲームに登場するオブジェクトを画面に置くことにする。このゲームでは、5 個の長方形が登場する。上下と右にある長方形は、ボールを跳ね返す壁である。左には 2 つの長方形が存在する。小さいほうはパドルである。ゲームをする人は、このパドルを上下に動かしてボールを打ち返す。大きいほうは、パドルでボールを打ち返せなかったことを知るための壁である。ボールがこの壁にぶつかると、ゲームオーバーになる。



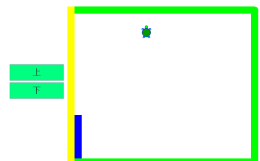
次のプログラムでは、5 個の長方形を作成している。プログラムを簡単にするために、**線の太さ**で太さ 20 の線を描くことで長方形とした。続いて、コの字の形に線を描き、「壁」という名前の図形を作っている。続いて上を向いた後、左壁とパドルを作っている。

このプログラムのように、座標を指定して図形やボタンを画面に置く場合には、終了ボタンの隣にある**方眼紙**をチェックして、位置を確認しながらプログラムを作ると便利である。

```
// 壁を作る (ステップ1)
カメ太=タートル! 作る。
カメ太! (緑) 線の色 20 線の太さ。
カメ太! 500 歩く 90 右回り 420 歩く 90 右回り 500 歩く。
壁=カメ太! 図形を作る -200 200 位置。
カメ太! 90 右回り。
左壁=カメ太! (黄) 線の色 440 歩く 図形を作る -210 -230 位置。
パドル=カメ太! (青) 線の色 120 歩く 図形を作る -190 -210 位置。
```

## 6.3 パドルを動かす (ステップ 2)

画面にボタンを表示して、パドルを上下に動かしてみよう。次のプログラムでは、画面の左側に「上ボタン」と「下ボタン」という名前の 2 つのボタンを表示して、それぞれが押されたときにパドルを上下に 50 ずつ移動させている。今回はボタンの 2 個目のパラメータに **UP** と **DOWN** を指定することで、上下の矢印キーでも操作できるようにした。



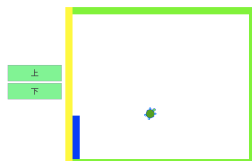
```
// パドルを動かす (ステップ2)
上ボタン=ボタン! "上" "UP" 作る -380 50 位置。
下ボタン=ボタン! "下" "DOWN" 作る -380 0 位置。
上ボタン:動作=「パドル! 0 50 移動する」。
下ボタン:動作=「パドル! 0 -50 移動する」。
```

## 6.4 ボールを動かす (ステップ 3)

「カメ太」には、壁やパドルを描いた後で、ボールの役割をしてもらうことにする。次のプログラムでは、カメ太が移動したときに線を引かないように、**ペンなし**を実行している。ボールは、最初は右の壁に向かって動くようにした。タートルの初期位置は、**乱数**を使い、少しずつ異なる位置から動き始めるようにした。横の位置 (x座標) は「乱数 (200)」で 1 から 200 の値に、縦の位置 (y座標) は「乱数 (300) - 150」で -149 から 150 の値になる。タートルの初期角度は 45 度で固定とした。

タートルが壁やパドルに衝突したときは、跳ね返る動作をすることが望ましい。そこで、カメ太の**衝突**には、タートルオブジェクトの**跳ね返る**を設定した。タートルのぶつかった向きに合わせて、自然な角度で跳ね返ることができる。

最後にタイマーを生成し、「カメ太! 20 歩く」を 0.1 秒間隔で 60 秒間繰り返し実行するようにした。実行すると、カメ太が壁で囲まれた空間を跳ね返りながら動き回る。



```
// ボールを動かす (ステップ3)
カメ太! ペンなし。
カメ太! (乱数 (200)) (乱数 (300) -150) 位置。
カメ太! 45 向き。
カメ太:衝突=タートル:跳ね返る。
時計=タイマー! 作る 60秒 時間「カメ太! 20 歩く」実行。
```

## 6.5 ゲームの勝敗を判定する (ステップ 4)

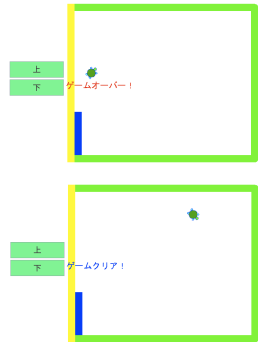
このゲームでは、パドルでボールを打ち返せずに 1 回でも左壁にボールが衝突したら負け (ゲームオーバー) とする。そして、タイマーでボールが動いている時間内にボールが左壁に衝

突しなければ勝ち（ゲームクリア）である。

次のプログラムでは、勝敗を判定するために、「ゲームクリア」という変数を導入した。この変数には**はい**または**いいえ**という真偽値を代入して使う。初期値は「はい」であるが、ボールが左壁に衝突したときは「いいえ」を代入する。このとき、「ゲームクリア」をルートのプロパティとして扱うために、前に「:」を付けている。そして、タイマーの実行を中断している。

タイマーの終了は「時計！ 待つ」で待つ。タイマーがどのようにに終了したのかは、「ゲームクリア」の値で判定している。値が真（はい）の場合は、無事に 60 秒間経って終了したので、青い字で「ゲームクリア」を表示している。値が偽（いいえ）の場合は、左壁に衝突して中断したので、赤い字で「ゲームオーバー」を表示するようにした。

```
// ゲームの勝敗を判定する（ステップ4）
ゲームクリア=はい。
左壁:衝突=「:ゲームクリア=いいえ。時計！ 中断」。
時計！ 待つ。
「ゲームクリア==はい」！ なら「
    ラベル！ "ゲームクリア！ "作る （青）文字色。
」そうでなければ「
    ラベル！ "ゲームオーバー！ "作る （赤）文字色。
」実行。
```



以上でピンポンゲームは完成である。実行すると、左壁に衝突したときは「ゲームオーバー！」が表示される。時間内にすべてのボールをパドルで打ち返せたときは、「ゲームクリア！」が表示される。

最後にステップ1からステップ4までの全体のプログラムを掲載しておく。

// 壁を作る (ステップ1)

カメ太=タートル! 作る。

カメ太! (緑) 線の色 20 線の太さ。

カメ太! 500 歩く 90 右回り 420 歩く 90 右回り 500 歩く。

壁=カメ太! 図形を作る -200 200 位置。

カメ太! 90 右回り。

左壁=カメ太! (黄) 線の色 440 歩く 図形を作る -210 -230 位置。

パドル=カメ太! (青) 線の色 120 歩く 図形を作る -190 -210 位置。

// パドルを動かす (ステップ2)

上ボタン=ボタン! "上" "UP" 作る -380 50 位置。

下ボタン=ボタン! "下" "DOWN" 作る -380 0 位置。

上ボタン: 動作=「パドル! 0 50 移動する」。

下ボタン: 動作=「パドル! 0 -50 移動する」。

// ボールを動かす (ステップ3)

カメ太! ペンなし。

カメ太! (乱数 (200)) (乱数 (300) -150) 位置。

カメ太! 45 向き。

カメ太: 衝突=タートル: 跳ね返る。

時計=タイマー! 作る 60秒 時間「カメ太! 20 歩く」実行。

// ゲームの勝敗を判定する (ステップ4)

ゲームクリア=はい。

左壁: 衝突=「: ゲームクリア=いいえ。時計! 中断」。

時計! 待つ。

「ゲームクリア==はい」! なら「

ラベル! "ゲームクリア! "作る (青) 文字色。

」そうでなければ「

ラベル! "ゲームオーバー! "作る (赤) 文字色。

」実行。

今後の改良としては、難易度を選べるようにして、実行時間、ボールの速さ、パドルの大きさなどを変えられるようにしてもよいかもしれない。また、パドルで打ち返した回数をカウントし、得点として表示することも考えられる。

## Activity 7



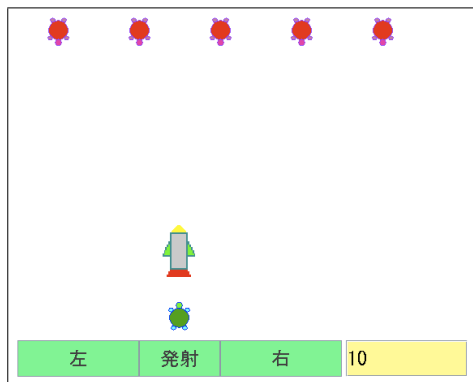
# シューティングゲーム

今まで学んだことを応用して、この Activity ではシューティングゲームを作る。シューティングゲームの作り方を理解しておく、さまざまなゲームに発展させて楽しむことができる。

ただし、いくつか難易度の高い点がある。解説をよく読んで、プログラムと見比べながら理解してほしい。プログラムは、ステップ1からステップ6までに分かれている。順に入力しながら動作を確認してほしい。全体のプログラムはステップ6の節で見ることができる。

## 7.1 作成するゲーム

画面の下部には主役のカメがいる。ボタンで左右に操作する。画面の上部には敵がいる。主役のカメからロケットを発射して攻撃することができる。敵に命中すると、画面下の得点欄に得点が加算されて行く。制限時間内にすべての敵を破壊することができるクリアとなる。

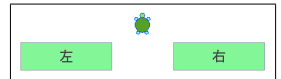


## 7.2 主役を作る (ステップ 1)

最初に、主役のオブジェクトを画面に置くことにする。このゲームでは、画面の下にいる「カメ太」が、上から降りてくる敵を攻撃する。まず、カメ太を作り、ボタンで左右に動かせるようにしてみよう。

次のプログラムでは、「カメ太」という名前のタートルオブジェクトを生成し、「90 左回り」で上向きにし、「ペンなし」で線を描かないように設定した後、「0 150 位置」で画面の下の中央付近に置いている。続いて、「左」と「右」という名前のボタンオブジェクトを作り、カメ太の下に置いた後、押されたときにカメ太の位置を  $x$  方向に 20 または  $-20$  だけ動かすように設定している。ボタンには **LEFT**、**RIGHT** を定義しているため、キーボードの左右の矢印キー（「←」、「→」）で操作することもできる。

プログラムを実行すると、画面にタートルとボタンが表示され、ボタンを押すことでタートルを左右に動かすことができる。



```
// 主役と左右の動き (ステップ1)
カメ太=タートル! 作る 90 左回り ペンなし 0 -150 位置。
左=ボタン! "左" "LEFT" 作る -200 -180 位置。
左:動作=「カメ太! -20 0 移動する」。
右=ボタン! "右" "RIGHT" 作る 50 -180 位置。
右:動作=「カメ太! 20 0 移動する」。
```

## 7.3 弾を発射する (ステップ 2)

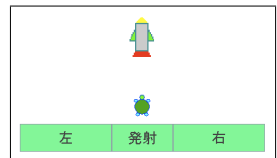
弾を発射するには、どのようなプログラムを書けばよいだろうか。弾は、「カメ太」がいる位置から出て行く必要がある。そして、敵に向かって進んでいく。

ここでは、「カメ太」の分身を作り、弾として前進させることにした。次のプログラムでは、「カメ太」に「発射」というメソッドを定義し、その中でタイマーオブジェクトを作り、自分を 20 歩ずつ前進させるようにしている。タイマーの実行間

隔は標準 (0.1 秒間隔) である。続いて、「発射」という名前のボタンオブジェクトを作り、左右のボタンの間に置いた後、押されたときに「カメ太」を「作る」で複製し、55 歩前進させた後、「rocket.gif」変身する」でロケットの姿にして、先ほど作った「発射」を実行させている。ボタンには **UP** を定義しているため、キーボードの上向きの矢印キー「↑」で操作することもできる。

ここで、最初に 55 歩前進している理由は、後から「衝突」を定義したときに、「カメ太」と弾が衝突してお互いが消えてしまう事故を防ぐためである。タートルの変身前の姿は半径が 20 程度であるため、55 歩離れてしまえば衝突は発生しない。

カメ太の「発射」メソッドでは、タイマーオブジェクトを作った後、続けてブロックを与えて実行している。このように、メッセージの**カスケード**を使うことで、プログラムを簡潔に記述している。



```
// 発射ボタン (ステップ2)
```

```
カメ太：発射＝「タイマー！ 作る「自分！ 20 歩く」実行」。
```

```
発射=ボタン！ "発射" "UP" 作る -50 -180 位置 100 45 大きさ。
```

```
発射:動作＝「カメ太！ 作る 55 歩く "rocket.gif" 変身する 発射」。
```

## 7.4 敵たちを作る (ステップ 3)

画面の上部に、敵を配置する。敵は横一列に並ぶようにした。後で敵をまとめて動かしたり、敵の残っている数を管理したりするために、敵を配列で管理する。

次のプログラムでは、最初に「敵たち」という名前の配列を作っている。続いて、「敵」という名前のタートルオブジェクトを作り、「ayumiAka.gif」変身する」で赤いタートルに変身させた後、「ペンなし」で線を描かないようにして、「90 右回り」で下を向かせている。そして次の行で、(-300, 200) の位置に移動させた後、配列「敵たち」に格納している。続く 4 行では、敵を「作る」で複製した後、それぞれを画面に配置してから配列「敵たち」に格納している。

最後の 4 行では、プログラムを簡潔に書くために、「敵」オブジェクトの複製と画面への配置、配列への格納を 1 行ずつ記

述した。もしプログラムが分かりづらければ、次のように、2行ずつに分けて記述することもできる。

```
// 1行で書いた例
敵たち！（敵！ 作る -200 200 位置）書く。
```

```
// 2行で書いた例
敵2=敵！ 作る -200 200 位置。
敵たち！（敵2）書く。
```



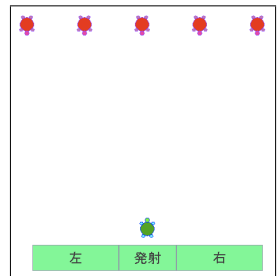
```
// 敵たちの生成（ステップ3）
敵たち=配列！ 作る。
敵=タートル！ 作る "ayumiAka.gif" 変身する ペンなし 90 右回り。
敵たち！（敵！ -300 200 位置）書く。
敵たち！（敵！ 作る -200 200 位置）書く。
敵たち！（敵！ 作る -100 200 位置）書く。
敵たち！（敵！ 作る 0 200 位置）書く。
敵たち！（敵！ 作る 100 200 位置）書く。
```

## 7.5 敵たちの移動（ステップ4）

敵は、1列に横に並んだまま、少しずつ右または左に動き、画面の端まで動くと少し下に進んで、先ほどとは逆の向きに横に移動するようにした。

次のプログラムでは、最初に「時計」という名前のタイマーを作り、実行間隔を1秒に設定している。続く3行では、タイマーを使って敵を移動する。「時計」という1つのタイマーオブジェクトに続けて「実行」を送ることで、「ひとつの実行が終わったら次を実行」という形で、**タイマーの逐次実行**を実現している。

最後の3行では、プログラムを簡潔に書くために、タイマーの実行と配列内の各オブジェクトの実行を1行に記述した。もしプログラムが分かりづらければ、次のように、2行ずつに分けて記述することもできる。ここで「右移動」はタイマーによって実行されるブロックなので、全体を括弧（「...」）で囲む必要がある。





```
// 1行で書いた例
```

```
時計!6 回数「敵たち！」「|敵|敵！ 30 0 移動する」それぞれ実行」実行。
```

```
// 2行で書いた例
```

```
右移動＝「敵たち！」「|敵|敵！ 30 0 移動する」それぞれ実行」。
```

```
時計!6 回数（右移動）実行。
```

```
// 敵たちの移動（ステップ4）
```

```
時計＝タイマー！ 作る 1秒 間隔 。
```

```
時計!6 回数「敵たち！」「|敵|敵！ 30 0 移動する」それぞれ実行」実行。
```

```
時計!1 回数「敵たち！」「|敵|敵！ 0 -30 移動する」それぞれ実行」実行。
```

```
時計!6 回数「敵たち！」「|敵|敵！ -30 0 移動する」それぞれ実行」実行。
```

## 7.6 衝突の定義 (ステップ 5)

弾が当たったときに、敵と弾が消えるようにする。敵が消えるときは、敵を管理するための配列「敵たち」からも削除する。

次のプログラムでは、最初に「カメ太」に「衝突」を定義して、何かにぶつかった場合に自分を消すようにしている。実際には弾がこの衝突を実行するが、「カメ太」に定義することによって、「カメ太」を複製して作られた弾にも自動的に定義されることになる。

続く 3 行では、配列「敵たち」に含まれるすべての敵オブジェクトに衝突を定義している。敵は何かにぶつかった場合に、自分を消してから、配列「敵たち」から自分を削除する。

```
// 衝突定義（ステップ5）
```

```
カメ太：衝突＝「自分！ 消える」。
```

```
敵：衝突＝「自分！ 消える。敵たち！（自分）消す」。
```

## 7.7 終了判定 (ステップ 6)

ゲームのプログラムでは、最後に定められた条件をクリアしたかどうかの判定が行われる。このゲームでは、時間内にすべての敵を消したかどうかを判定することにした。

次のプログラムでは、制限時間を 15 秒にするために、「制限時間」という変数を作り 15 を代入している。続いて、残り時

間を表示するために、「カウントダウン」という名前のフィールドを作っている。続いて、「終了時計」という名前のタイマーオブジェクトを作り、1秒間隔で「制限時間」だけ動くようにした。

続く6行は、「終了時計」の実行内容であり、1秒間隔で「制限時間」の秒数だけ実行される。何回目の実行かは、先頭のパラメータ「n」に入る。フィールド「カウントダウン」には、「制限時間 - n」という式で、残り時間を表示するようにした。続いて、配列「敵たち」の要素数を調べ、値がゼロであれば、ラベルで「おめでとう!!!」というメッセージを表示する。そして、終了時計を中断してカウントダウンを止める。

このプログラムでは、制限時間内にクリアしたときにメッセージを表示している。もしクリアできなかったときに別のメッセージを表示したい場合は、終了時計の終了を「待つ」で待ち、そのときの「敵たち」の要素数を調べればよい。

```
// 終了判定 (ステップ6)
```

```
制限時間=15。
```

```
カウントダウン=フィールド！ 作る 150 0 移動する。
```

```
終了時計=タイマー！ 作る 1秒 間隔（制限時間）時間。
```

```
終了時計！「|n|カウントダウン！（制限時間-n）書く。
```

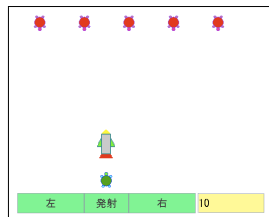
```
「(敵たち！ 要素数?) == 0」！ なら「
```

```
ラベル！ "おめでとう!!! "作る -100 200 位置。
```

```
終了時計！ 中断
```

```
」実行
```

```
」実行。
```



以上でシューティングゲームは完成である。

最後にステップ1からステップ6までの全体のプログラムを掲載しておく。

// 主役と左右の動き (ステップ1)

カメ太=タートル! 作る 90 左回り ペンなし 0 -150 位置。

左=ボタン! "左" "LEFT" 作る -200 -180 位置。

左:動作=「カメ太! -20 0 移動する」。

右=ボタン! "右" "RIGHT" 作る 50 -180 位置。

右:動作=「カメ太! 20 0 移動する」。

// 発射ボタン (ステップ2)

カメ太:発射=「タイマー! 作る「自分! 20 歩く」実行」。

発射=ボタン! "発射" "UP" 作る -50 -180 位置 100 45 大きさ。

発射:動作=「カメ太! 作る 55 歩く "rocket.gif" 変身する 発射」。

// 敵たちの生成 (ステップ3)

敵たち=配列! 作る。

敵=タートル! 作る "ayumiAka.gif" 変身する ペンなし 90 右回り。

敵たち! (敵! -300 200 位置) 書く。

敵たち! (敵! 作る -200 200 位置) 書く。

敵たち! (敵! 作る -100 200 位置) 書く。

敵たち! (敵! 作る 0 200 位置) 書く。

敵たち! (敵! 作る 100 200 位置) 書く。

// 敵たちの移動 (ステップ4)

時計=タイマー! 作る 1秒 間隔。

時計!6 回数「敵たち! 「|敵|敵! 30 0 移動する」それぞれ実行」実行。

時計!1 回数「敵たち! 「|敵|敵! 0 -30 移動する」それぞれ実行」実行。

時計!6 回数「敵たち! 「|敵|敵! -30 0 移動する」それぞれ実行」実行。

// 衝突定義 (ステップ5)

カメ太:衝突=「自分! 消える」。

敵:衝突=「自分! 消える。敵たち! (自分) 消す」。

// 終了判定 (ステップ6)

制限時間=15。

カウントダウン=フィールド! 作る 150 0 移動する。

終了時計=タイマー! 作る 1秒 間隔 (制限時間) 時間。

終了時計! 「|n|カウントダウン! (制限時間-n) 書く。

「(敵たち! 要素数?) == 0」! なら「

ラベル! "おめでとう!!! "作る -100 200 位置。

終了時計! 中断

」実行

」実行。



Part IV

# 音楽を演奏しよう



## Activity 8



# 音楽の演奏

ドリトルでは **MIDI 音源**<sup>\*1</sup> を使った楽器演奏を行える。演奏する**メロディ**は、「どれみー」といった分かりやすい形で記述する。**コード**による伴奏と**ドラム**による打楽器演奏を合わせて、**バンド**や**オーケストラ**のような合奏が可能である。

## 8.1 メロディの演奏

メロディを演奏するには、譜面に相当する**メロディオブジェクト**に**旋律**を書き込み、それを演奏する。次のプログラムでは、「きらきらぼし」という名前のメロディオブジェクトを生成し、「"...」の形で旋律を**追加**してから、メロディオブジェクトに**演奏**させている。音符を記述する「"...」の部分は、「『...』」という記号で囲んでもよい。楽譜に演奏を指示すると、標準の楽器である**ピアノ**を使った演奏が行われる。

きらきらぼし＝メロディ！ 作る。

きらきらぼし！ "ドドソソララソ～ファファミミレド～" 追加。

きらきらぼし！ "ソソファファミレ～ソソファファミレ～" 追加。

きらきらぼし！ "ドドソソララソ～ファファミミレド～" 追加。

きらきらぼし！ 演奏。

## 8.2 メロディの記述

メロディには、「ドレミ」といった音階に加えて、数字や記号で細かい指定を行える。

<sup>\*1</sup> MIDI はコンピュータで音楽を演奏するための規格。

オクターブ上に移るときは「`^`」を、オクターブ下に移るときは「`_`」を指定する。ひとつの音だけでなく、それ以降がすべてオクターブ上（または下）になる。音を伸ばすときは「`~`」や「`-`」で指定する。音の後ろに 1, 2, 4, 8, 16 といった数字を書くことで、それぞれ全音符、二分音符、四分音符、八分音符、十六分音符を指定することもできる。

次のプログラムは、これらを使ったサンプルである。（実行しない行は「`//`」で一時的にコメントにしている）

```
きらきらばし=メロディ！ 作る。
きらきらばし！ "ド・ドソソソソソソ~_ファファ1ミ2ミ4レレ8ド~" 追加。
// きらきらばし！ "ソソファファミミレ~ソソファファミミレ~" 追加。
// きらきらばし！ "ドドソソソソソソ~ファファミミレレド~" 追加。
きらきらばし！ 演奏。
```

## 8.3 楽器の指定

標準のピアノ以外に、いくつかの楽器を指定して演奏できる。あらかじめ、いくつかの楽器が用意されている（E.6 節）。次のプログラムでは、楽器をオルガンに設定して演奏している。

```
きらきらばし=メロディ！ 作る。
きらきらばし！ "ドドソソソソソソ~ファファミミレレド~" 追加。
きらきらばし！ "ソソファファミミレ~ソソファファミミレ~" 追加。
きらきらばし！ "ドドソソソソソソ~ファファミミレレド~" 追加。
きらきらばし！ (楽器！ "オルガン" 作る) 設定。
きらきらばし！ 演奏。
```

次のプログラムでは、「メロディ欄」という名前の**フィールド**オブジェクトを作り、そこに初期値として「ドドソソソソソソ~ファファミミレレド~」という旋律を入れている。この旋律は実行中にキーボードから修正することができる。続いて「楽器名」という名前の**選択メニュー**オブジェクトを作り、ピアノ、オルガン、ギター、トランペット、ベルを設定している。最後に「実行ボタン」という名前の**ボタン**オブジェクトを作り、ボタンを押したときに「メロディ欄」に書かれた旋律を「楽器名」で指定された楽器で演奏する。

ドドソソララソ〜ファファミミレレド〜

ピアノ

実行

ピアノ

オルガン

ギター

トランペット

ベル

メロディ欄=フィールド！ 作る 600 45 大きさ。

メロディ欄！ "ドドソソララソ〜ファファミミレレド〜" 書く。

楽器選択=選択メニュー！ "ピアノ" "オルガン" "ギター" "トランペット" "ベル" 作る 次の行。

バンド1=バンド！ 作る。

楽器選択：動作=「 | 楽器名 |

選んだ楽器=楽器！（楽器名）作る。

主旋律=メロディ！ 作る（メロディ欄！ 読む）追加（選んだ楽器）設定。

主旋律！ 演奏。

」。

## 8.4 音楽の構造をプログラムする

「きらきらぼし」のメロディをもう一度見てみよう。

きらきらぼし=メロディ！ 作る。

きらきらぼし！ "ドドソソララソ〜ファファミミレレド〜" 追加。

きらきらぼし！ "ソソファファミミレ〜ソソファファミミレ〜" 追加。

きらきらぼし！ "ドドソソララソ〜ファファミミレレド〜" 追加。

きらきらぼし！ 演奏。

メロディの部分をよく見ると、次のような**曲の構造**があることが分かる。

- 「ドドソソララソ〜ファファミミレレド〜」 という同じ旋律が、最初と最後に出てきている。（A メロディと呼ぶことにする）
- 「ソソファファミミレ〜ソソファファミミレ〜」 という2番目の旋律は、「ソソファファミミレ〜」 が2回繰り返されている。（B メロディと呼ぶことにする）



これを整理すると、「きらきらぼし」という曲は、次のような構造を持っていることが分かる。

```
Aメロディ
Bメロディが2回
Aメロディ
```

分かりやすいように、プログラムの最初で「Aメロディ」と「Bメロディ」という**変数**を定義すると、次のように書ける。

```
// 変数の定義
Aメロディ="ドドソソラソ〜ファファミミレド〜"。
Bメロディ="ソソファファミミレ〜"。

// 演奏
きらきらぼし=メロディ！ 作る。
きらきらぼし！（Aメロディ）追加。
きらきらぼし！（Bメロディ）追加（Bメロディ）追加。
きらきらぼし！（Aメロディ）追加。
きらきらぼし！ 演奏。
```

2 個の「Bメロディ」を**繰り返し**で書くと、次のようになる。

```
// 変数の定義
Aメロディ="ドドソソラソ〜ファファミミレド〜"。
Bメロディ="ソソファファミミレ〜"。

// 繰り返しを使った演奏
きらきらぼし=メロディ！ 作る。
きらきらぼし！（Aメロディ）追加。
「きらきらぼし！（Bメロディ）追加」！ 2回 繰り返す。
きらきらぼし！（Aメロディ）追加。
きらきらぼし！ 演奏。
```

このプログラムを、「Aメロディを2回繰り返す中で、1回目のときだけAメロディにBメロディを2個追加する」という方針で書き換えると、次のようになる。Aメロディの繰り返しでは、**ブロック**(「...」)の先頭で「Aメロ回数」という**パラメータ**で、繰り返しの回数を受け取っている。そして、「Aメロ回数==1」という**条件式**で、1回目の繰り返しかどうかを判断している。

```
// 変数の定義
```

```
Aメロディ="ドドソソララソ～ファファミミレド～"。
```

```
Bメロディ="ソソファファミミレ～"。
```

```
きらきらぼし=メロディ！ 作る。
```

```
// 繰り返しと条件分岐を使った演奏
```

```
「|Aメロ回数|
```

```
    きらきらぼし！（Aメロディ）追加。
```

```
    「Aメロ回数==1」！ なら「
```

```
        「きらきらぼし！（Bメロディ）追加」！ 2回 繰り返す。
```

```
    」実行。
```

```
」！ 2回 繰り返す。
```

```
きらきらぼし！ 演奏。
```

この例で見たように、多くの曲では、ひとつの曲の中にいくつかのメロディの断片が繰り返して含まれている。音楽をプログラムで演奏する際には、単に楽譜を入力するのではなく、曲の構造をうまく見つけてプログラムで表現することが大切である。

## 8.5 メロディの合奏

同じメロディを何拍かずらして演奏することで、輪唱を演奏できる。次のプログラムでは、「かえる 2」というメロディの先頭に 8 拍の無音を入れ、続けて「かえる 1」というメロディを加えている。そして、「かえる 1」と「かえる 2」を同時に演奏している。

バンドオブジェクトを使うと、複数のメロディを演奏することができる。次のプログラムでは、「輪唱」というバンドオブジェクトを作り、バンドのメンバーとして、「かえる 1」と「かえる 2」を追加している。輪唱を演奏すると、「かえる 1」と「かえる 2」が同時に演奏される。

```
かえる1=メロディ！ 作る。
```

```
かえる1！ "ドレミファミレド～ミファソラソファミ～" 追加。
```

```
かえる1！ "ド・ド・ド・ド・ド8ド8レ8レ8ミ8ミ8ファ8ファ8ミレド～" 追加。
```

```
かえる2=メロディ！ 作る 8 無音 （かえる1）追加。
```

```
輪唱=バンド！ 作る （かえる1）追加 （かえる2）追加。
```

```
輪唱！ 演奏。
```

## 8.6 楽器を変えて演奏する

標準の楽器であるピアノ以外にも楽器が用意されている。先ほどの輪唱プログラムは、ピアノ 2 台だったので 2 つのメロディの区別がつきにくかった。ここでは 1 台の楽器を**オルガン**に変えて演奏してみる。次のプログラムでは、「オルガン 1」という名前の**楽器**オブジェクトを作り、「かえる 2」に**設定**して演奏している。

```
かえる1=メロディ！ 作る。
かえる1！ "ドレミファミレド～ミファソラソファミ～" 追加。
かえる1！ "ド・ド・ド・ド・ド8ド8レ8レ8ミ8ミ8ファ8ファ8ミレド～" 追加。
かえる2=メロディ！ 作る 8 無音 （かえる1）追加。
オルガン1=楽器！ "オルガン" 作る。
かえる2！（オルガン1）設定。
輪唱=バンド！ 作る （かえる1）追加 （かえる2）追加。
輪唱！ 演奏。
```

楽器は 1～128 で表される MIDI の楽器番号<sup>\*2</sup> で指定することもできる。次のプログラムでは、**乱数**で楽器を指定している。実行するたびに、異なる楽器で演奏される。乱数のような数式は、括弧 (...) で囲んで記述する。

```
かえる=メロディ！ 作る。
かえる！ "ドレミファミレド～ミファソラソファミ～" 追加。
私の楽器=楽器！（乱数（128））作る。
かえる！（私の楽器）設定。
かえる！ 演奏。
```

## 8.7 楽譜からの入力

音楽の楽譜を見ながら、曲を入力してみよう。

<sup>\*2</sup> 付録 E を参照。

## 自分で楽譜を入力するために

### 楽譜と音階 (基本) ト音記号の音階



音階のオクターブの境目(下の楽譜の↓のところ)では、  
オクターブを上げる ( ^ )、下げる ( \_ ) といった命令が必要



\_ (アンダーバー) や ^ (ハットマーク) は音階の前に付けます。

また、1オクターブ上や下の音が連続するときは、一回だけ指定します。(命令が変わると指定が必要です)

### オクターブを越える例



# (半音あがる) や ♭ (半音下がる) がついた音は、音階のあとに#や♭をつけます。



楽譜の最初に調号があるときは、それに対応した音階に#や♭をつけます。



音の長ささと命令 音階のあとに続けます。(4については省略できます)

(16分音符)	(8分音符)	(4分音符)	(2分音符)	(全音符)
ド16 ド16 ド16	ド8 ド8 ド8	ド4	ド2	ド1

三連符(音符2つ分の長さに3つの音符が入るもの)の場合は、音を続けて { } で囲みます。

{ドドド}16 {ドレミ}16	{ドドド}8 {ドレミ}8	{ドドド}4 {ドレミ}4

付点は長さが半分の音を追加します。(右と左は同じ長さです)

ド4. ド4 & 8	ド2. ド2 & 4	ド1. ド1 & 2

休符の長ささと命令 休符は・であらわします。

(16分休符)	(8分休符)	(4分休符)	(2分休符)	(全休符)
・16	・8	・4	・2	・1

ヘ音記号の音階

—ド レ ミ ファ ソ ラ シ	

実際の楽器で演奏するとき、ヘ音記号の音の高さはト音記号の音より1オクターブ低い音になります。  
メロディがト音記号の音階、伴奏がヘ音記号の音階の楽譜を入力する場合、ヘ音記号の音階の方には、—をつけておきましょう。

この2つは同じ高さの音

## 8.8 リズム

打楽器のリズムを演奏するには、**ドラム**オブジェクトを使用する。使い方はメロディオブジェクトなどと同様である。音階の代りに、「ドツタツドツタツ」といったリズムを記述する。

次のプログラムでは、最初に「きらきらぼし」という名前のメロディオブジェクトを作り、続いて「きらりずむ」という名前のドラムオブジェクトを作っている。「ドツタツ」といった表記は、「ド」がバスドラ、「ツ」がハイハットなど、それぞれがドラムで使われる楽器を表す。メロディと違って、音の長さは半拍である。詳しくは付録 E を参照してほしい。最後にバンドオブジェクトを作り、「きらきらぼし」と「きらりずむ」を追加して演奏している。

きらきらぼし=メロディ!"ドドソソララソ〜ファファミミレド〜" 作る。  
 きらりずむ=ドラム！ "ドツタツドツタツドツタツドタタツドツタツドツタツドツ  
 タツクチパン" 作る。  
 バンド！ 作る（きらきらぼし）追加（きらりずむ）追加 60 テンポ 演奏。

もちろん、バンドに名前を付けることもできる。次のプログラムでは「マイバンド」という名前のバンドを作り演奏している。

きらきらぼし=メロディ!"ドドソソララソ〜ファファミミレド〜" 作る。  
 きらりずむ=ドラム！ "ドツタツドツタツドツタツドタタツドツタツドツタツドツ  
 タツクチパン" 作る。  
 マイバンド=バンド！ 作る。  
 マイバンド！（きらきらぼし）追加（きらりずむ）追加。  
 マイバンド！ 60 テンポ 演奏。

## Activity 9



# 音楽で楽しもう

## 9.1 琉球音階の自動作曲

沖縄の音階（琉球音階）を使って、プログラムで作曲してみよう。

琉球音階は、「ドレミファソラシ」の「レ」と「ラ」が存在しない音階である。次のプログラムでは、「琉球音階」という**配列**に、基本となる音階を入れた後で、それぞれの音をメロディオブジェクトに入れて演奏している。

琉球音階＝配列！ "ド" "ミ" "ファ" "ソ" "シ" 作る。

僕の楽譜＝メロディ！ 作る。

僕の楽譜！（琉球音階！ 1 読む）追加。

僕の楽譜！（琉球音階！ 2 読む）追加。

僕の楽譜！（琉球音階！ 3 読む）追加。

僕の楽譜！（琉球音階！ 4 読む）追加。

僕の楽譜！（琉球音階！ 5 読む）追加。

僕の楽譜！ 演奏。

このプログラムは繰り返しを使って書くこともできる。ここでは、何回目の繰り返しかを**ブロック**（「...」）のパラメータ「n」として受け取り、その番号の音を取り出してメロディに追加している。

琉球音階＝配列！ "ド" "ミ" "ファ" "ソ" "シ" 作る。

僕の楽譜＝メロディ！ 作る。

「n」僕の楽譜！（琉球音階！（n）読む）追加！ 5回 繰り返す。

僕の楽譜！ 演奏。

配列である琉球音階からは、5種類の音を1から5の数字で取り出せる。乱数で1～5の数を発生させることで、ランダムな音を演奏することも可能である。

琉球音階＝配列！ "ド" "ミ" "ファ" "ソ" "シ" 作る。

僕の楽譜＝メロディ！ 作る。

「僕の楽譜！（琉球音階！（乱数（5）読む）追加）！ 5回 繰り返す。

僕の楽譜！ 演奏。

次のプログラムでは、乱数でAメロディとBメロディという短い旋律を作り、それらを組み合わせて曲を作っている。Aメロディ6個の音を、Bメロディは8個の音を、それぞれ乱数でつなげて作っている。そして、Aメロディの最後には「ド～」を加えている。曲の構成としては、Aメロディを2回演奏し、Bメロディを演奏した後、最後にAメロディをもういちど演奏して終る。楽器には「ギター」を設定した。

琉球音階＝配列！ "ド" "ミ" "ファ" "ソ" "シ" 作る。

Aメロディ＝メロディ！ 作る。

「Aメロディ！（琉球音階！（乱数（5）読む）追加）！ 6回 繰り返す。

Aメロディ！ "ド～" 追加。

Bメロディ＝メロディ！ 作る。

「Bメロディ！（琉球音階！（乱数（5）読む）追加）！ 8回 繰り返す。

僕の楽譜＝メロディ！ 作る。

「僕の楽譜！（Aメロディ）追加」！ 2回 繰り返す。

僕の楽譜！（Bメロディ）追加。

僕の楽譜！（Aメロディ）追加。

僕の楽器＝楽器！ "ギター" 作る。

僕の楽譜！（僕の楽器）設定。

僕の楽譜！ 演奏。



## 9.2 ランダムな音階の自動作曲

今度は、一定範囲の音をランダムに作り出してみよう。ここでは、2 オクターブの音階の音をランダムに演奏することにする。

次のプログラムでは、最初に乱数で作る音の基準となる音程を決めている。ここでは「ド」にした。そして音程を書いていくためのメロディを作り、「楽譜」という名前にしている。続いて、基準の音から2 オクターブ分の音をランダムに作り、楽譜に50回追加している。「乱数 (24)」は1~24の値をランダムに返す。「音上げる」は指定された数だけ半音ずつ音程を上げる。最大24個の半音なので、2 オクターブ上の音階まで表現できることになる。最後に、作った楽譜を演奏している。

音程＝メロディ！ 作る "ド"追加。

楽譜＝メロディ！ 作る。

「楽譜！ （音程！ （乱数 (24)) 音上げる）追加」！ 50回 繰り返す。

楽譜！ 演奏。

## 9.3 フレーズを利用した自動作曲

配列に複数の音を入れて演奏してみよう。このプログラムは、複数パートの曲を作る場合にも応用できる。

次のプログラムでは、最初にメロディの元になるフレーズを配列に格納している。これらのフレーズは、演奏したときに同じ長さになるように揃えておく。次に、パートごとにメロディを作り、楽器を割り当てる。ここでは楽器番号で指定した。続いて、パートごとにランダムにフレーズをつなげてメロディーを作る。最後に各パートを集めてバンドを作り、テンポを決めて演奏している。

フレーズ＝配列！ "ドレミファミレド～" "ミファソラソファミ～" "ド・ド・ド・ド・" "ド8ド8レ8レ8ミ8ミ8ファ8ファ8ミレド～" 作る。

パート1＝メロディ！ 作る（楽器！ 10 作る） 設定。

パート2＝メロディ！ 作る（楽器！ 11 作る） 設定。

パート3＝メロディ！ 作る（楽器！ 12 作る） 設定。

「パート1！（フレーズ！（乱数（4））読む）追加」！ 8 繰り返す。

「パート2！（フレーズ！（乱数（4））読む）追加」！ 8 繰り返す。

「パート3！（フレーズ！（乱数（4））読む）追加」！ 8 繰り返す。

合奏＝バンド！ 作る（パート1）追加（パート2）追加（パート3）追加（68）  
テンポ 演奏。

## 9.4 リズムと組み合わせた自動作曲

最後に、メロディとリズムを組み合わせた自動作曲をしてみよう。プログラムを分かりやすくするために、ここでは作曲する部分をメロディオブジェクトとドラムオブジェクトにメソッドとして定義している。作成する曲は、笛と太鼓の二重奏である。

次のプログラムでは、最初に笛の楽器番号と曲の速さ、音符の数の変数を定義している。このように、調整したい値を変数として定義しておくと、後から変更するときに便利である。続いて、笛と太鼓用の音階の配列を用意している。

作曲は、メロディとドラムに定義したメソッドで行う。笛のメロディは、音符として7個の音階をランダムに選び、その長さをランダムに決めたものを繰り返すようにした。「↑」は「^」と同じ意味で、続く音階が1オクターブ上がる。「↑↑」は2オクターブ、「↑↑↑」は3オクターブ上がることを示す。太鼓のリズムは、3個の打楽器をランダムに選び、その長さをランダムに決めている。太鼓ではリズムを細かく刻むために、音符の長さは笛の半分（8分音符）に設定し、音符の数を倍に設定した。最後に笛と太鼓でバンドを作り、指定した曲の速さで演奏している。

笛楽器=072。曲の速さ=120。音符の数=50。

笛の音=配列！ "↑↑レ" "↑↑ミ" "↑↑ファ" "↑↑ラ" "↑↑シ" "↑↑↑レ" "↑↑↑ミ" 作る。

太鼓の音=配列！ "ど" "た" "つ" 作る。

笛=メロディ！ 作る（楽器！（笛楽器）作る）設定。

太鼓=ドラム！ 作る。

笛：作曲=「|n|」

音符=笛の音！ （乱数（7））読む。

「乱数（2）>1」！ なら「自分！（音符 + "4"）追加」そうでなければ「自分！（音符 + "2"）追加」実行。

」！ （n）繰り返す」。

太鼓：作曲=「|n|」

音符=太鼓の音！ （乱数（3））読む。

「乱数（2）>1」！ なら「自分！（音符 + "8"）追加」そうでなければ「自分！（音符 + "4"）追加」実行。

」！ （n）繰り返す」。

笛パート=笛！ （音符の数）作曲。

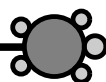
太鼓パート=太鼓！（音符の数 \* 2）作曲。

バンド！ 作る（笛パート）追加 （太鼓パート）追加 （曲の速さ）テンポ 演奏。



Part V

# ジェスチャーで操作 しよう



## Activity 10



# LeapMotion を使ってみよう

**LeapMotion** というデバイスを使って、手の動きでタートルなどを操作しよう。LeapMotion を使用するためには、ローカル版のドリトルが必要である。

LeapMotion は空中の手指の位置などを赤外線センサーで認識できる入力デバイスである。



## 10.1 ドリトルと通信するための設定

LeapMotion のサイトにアクセスする。<sup>\*1</sup> 「セットアップ」から「Windows 用ダウンロード」「Mac 用ダウンロード」などを選び、ダウンロードしたファイルから LeapMotion のソフトウェアをインストールする。

---

<sup>\*1</sup> <https://www.leapmotion.com/?lang=jp>

## 10.2 手の位置と指の情報の取得

動作を確認するには、コンピュータに LeapMotion を接続し、ドリトルを起動する。ドリトルに次のプログラムを入力して実行する。画面のタートルが LeapMotion の上の空間に置いた手の位置によって移動すれば、LeapMotion は正常に動作している。

このプログラムでは、手の横の位置でタートルの向きを変え、手の高さでタートルの歩く速度を変えている。

かめた＝タートル！ 作る。

リープ！ 接続。

時計＝タイマー！ 作る 600 時間。

時計！ 「

「(リープ！ 横の位置？) >50」！ なら「かめた！ 15 右回り。」実行。

「(リープ！ 横の位置？) <-50」！ なら「かめた！ 15 左回り。」実行。

「(リープ！ 縦の位置？) >100」！ なら「かめた！ ((リープ！ 縦の位置？) × 0.03) 歩く」実行。

」実行。

## 10.3 ジェスチャーの取得

LeapMotion は、手が特定の動きをしたときに、動きをジェスチャーとして認識できる。

このプログラムでは、指の本数を**グー**と**パー**で認識し、パーの場合は画面に星を増やし、グーの場合はそれらの星を移動させている。手を垂直に回転させると「回転」のジェスチャーと認識され画面の星が回転する。手を下に動かすと「キータップ」のジェスチャーと認識され画面の星が消える。

かめた＝タートル！ 作る。

かめた！ "star.png" 変身する ペンなし きえる。

かめたち＝配列！ 作る。

リープ！ 接続。

リープ：キータップ＝「かめたち！ 「|かめ| かめ！ 消える」それぞれ実行」。時計＝タイマー！ 作る 600 時間。

時計！ 「

幅＝画面！ 幅？。高さ＝画面！ 高さ？。

「リープ！ パー？」！ なら「かめたち！ （かめた！ 作る（乱数（幅）－（幅／2）） （乱数（高さ）－（高さ／2）） 位置）書く。」実行。

「リープ！ グー？」！ なら「かめたち！ 「|かめ| かめ！ 10 歩く」それぞれ実行」実行。

「リープ！ 右回転？」！ なら「かめたち！ 「|かめ| かめ！ 15 右回り」それぞれ実行」実行。

「リープ！ 左回転？」！ なら「かめたち！ 「|かめ| かめ！ 15 左回り」それぞれ実行」実行。

」実行。



Part VI

# センサやLEDと入出力しよう



## Activity 11

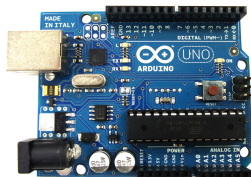


# Arduino と通信しよう

**Arduino** というデバイスを使って、LED やセンサなど外部デバイスとの入出力を試みよう。Arduino を使用するためには、ローカル版のドリトルが必要である。

## 11.1 Arduino の入手

Arduino は、イタリアで開発された汎用入出力ボードである。設計がオープンソースとして公開されており、さまざまな製品が存在する。<sup>\*1</sup>今回は、「Arduino UNO」という最新のボードで動作を確認した。このボードにはデジタル入出力が 12 ポート存在し、そのうち 6 ポートはアナログ出力が可能である。その他に、アナログ入力が 6 ポート用意されている。<sup>\*2</sup>



## 11.2 ドリトルと通信するための設定

Arduino は標準では、Processing 言語で記述されたプログラムを転送して自律的に動作する。本書では Arduino にドリトルと通信するためのプログラムを転送し、ドリトルと通信しながら対話的に動作させる。Arduino に格納したプログラムは電源を切っても残るため、いちど書きこんでおけばよい。以下に手順を示す。

<sup>\*1</sup> 入手方法は、Arduino の Web サイト (<http://www.arduino.cc/>) の「Buy」というページから「Japan」の項目で日本での代理店を見ることができる。

<sup>\*2</sup> Arduino UNO の基板上では、アナログ出力可能なポート番号には「〜」の記号が付いている。

1. Arduino の Web サイト (<http://www.arduino.cc/>) の「Download」ページから、使用しているコンピュータ (Windows, Mac OS X, Linux など) に合わせたソフトウェアをダウンロードし、インストールする。
2. インストールしたプログラム (**Arduino IDE**) を起動し、ドリトルのパッケージに付属する **arduino\_dolittle.ino** というファイルを開く。すると、画面に「`#include <LiquidCrystal.h>`」で始まるプログラムが表示される。
3. 「Tools」メニューの「Board」から、使用する Arduino の種類を選ぶ。
4. Arduino IDE 上部の「▷」のボタン (Verify) を押す。すると、画面下部に「Done Compiling」と「Binary sketch site: ...」というメッセージが表示される。
5. USB ケーブルで Arduino をパソコンに接続する<sup>\*3</sup>。すると、基板上の緑色の LED が点灯する。
6. Arduino IDE 上部の「⇒」のボタン (Upload) を押す。シリアルポートを選択するダイアログが開いた場合は、Arduino が接続されたポートを選択する<sup>\*4</sup>。すると、基板上のオレンジ色の LED が点滅してプログラムが転送される。
7. 正常に転送された場合は、Arduino IDE の画面下部に「Done uploading」と表示される。このメッセージを確認したら、Arduino IDE を終了する。

## 11.3 デジタル出力による動作の確認

動作を確認するには、Arduino と USB ケーブルを接続した状態で、ドリトルに次のプログラムを入力して実行する。Arduino 上の「L」と書かれたオレンジ色の LED が1秒ごとに

<sup>\*3</sup> Macintosh では設定が必要な場合がある。「はじめに」(p.iv)を参照。「新しいネットワークインターフェースが検出されました」と表示された場合は、キャンセルでダイアログを閉じて構わない。

<sup>\*4</sup> ポートが不明な場合は、Windows であれば「COM」に続く数字の大きい順に試し、Arduino が反応するポートを探す。Macintosh の場合は「dev/tty.usb」で始まる名前を探す。ポートは「Tools」メニューの「Serial Port」から選択することもできる。

10 回点滅すれば、ドリトルとの通信は正常に行われている。

```
システム！ "arduino"使う。
a=arduino！ 作る。
a！（システム！ シリアルポート選択）ひらけごま。

led=a！ 13 デジタル出力。
「led！ 0 書く。a！ 1 待つ。led！ 1 書く。a！ 1 待つ」！ 10 繰り返す。

a！ とじごま。
```

ドリトルから Arduino を制御するプログラムは、次の形で記述する\*5。先頭の行では、Arduino を**使う**プログラムを作成することを示している。2 行目の「a」は、接続された Arduino に対応するオブジェクトである。

```
システム！ "arduino"使う。
a=arduino！ 作る。
a！（システム！ シリアルポート選択）ひらけごま。
...
a！ とじごま。
```

「...」の部分には、Arduino を制御するプログラムを書く。今回は最初に、13 番ポートに**デジタル出力**するオブジェクトを作り「led」という名前を付けている。基板上には 13 番ポートに接続された**発光ダイオード（LED）**が存在するため、外部に LED を接続しなくても動作の確認が可能である。

```
led=a！ 13 デジタル出力。
```

続いて「led」に「書く」により出力を行う。デジタル出力では、「1」を書くことで出力を ON にし、「0」を書くことで出力を OFF にする。このプログラムでは、0 と 1 を 1 秒ごとに出力することで、LED を点滅させている。

```
「led！ 0 書く。a！ 1 待つ。led！ 1 書く。a！ 1
待つ」！ 10 繰り返す。
```

\*5 「（システム！ シリアルポート選択）」の部分は、あらかじめわかっている場合は、「COM1」のようにポート名を文字列で記述することも可能である。

Arduino に接続した LED を光らせる場合は、「デジタル出力」のパラメータに接続したポート番号を指定する。ブレッドボードを利用した配線の例は 11.7 を参照されたい。

## 11.4 アナログ出力

Arduino の特定のポート<sup>\*6</sup>では、PWM<sup>\*7</sup>による**アナログ出力**が可能である。アナログ出力では、出力する値として 0 から 255 の数値を指定できる。次のプログラムでは、最初に 10 番ポートに接続された LED を「150」の明るさで点灯させた後、「255」の明るさで点灯し、最後に「0」で消灯している。

```
システム！ "arduino"使う。
a=arduino！ 作る。
a！（システム！ シリアルポート選択）ひらけごま。

led=a！ 10 アナログ出力。
led！ 150 書く。a！ 1 待つ。
led！ 255 書く。a！ 1 待つ。
led！ 0 書く。

a！ とじごま。
```

## 11.5 デジタル入力

**デジタル入力**は、特定のポートの電圧の有無を 0 と 1 の値で返すことで、スイッチ等の ON と OFF を検出できる。

次のプログラムでは、12 番ポートに入力があるときに画面のタートルを前進させている。（スイッチが接続されている場合は、押されているかどうかでタートルの動きが変化する）

<sup>\*6</sup> Arduino UNO では基板上の番号の前に「-」が付いた 3, 5, 6, 9, 10, 11 の 6 個のポートでアナログ出力が可能である。

<sup>\*7</sup> ON と OFF の比率を変えながら高速に繰り返すことで、擬似的にアナログ値を表現する方式。

```
システム！ "arduino"使う。  
a=arduino！ 作る。  
a！（システム！ シリアルポート選択） ひらけごま。  
  
sw=a！ 12 デジタル入力。  
かめた=タートル！ 作る。  
「かめた！（sw！ 読む）歩く」！ 100 繰り返す。  
  
a！ とじごま。
```

## 11.6 アナログ入力

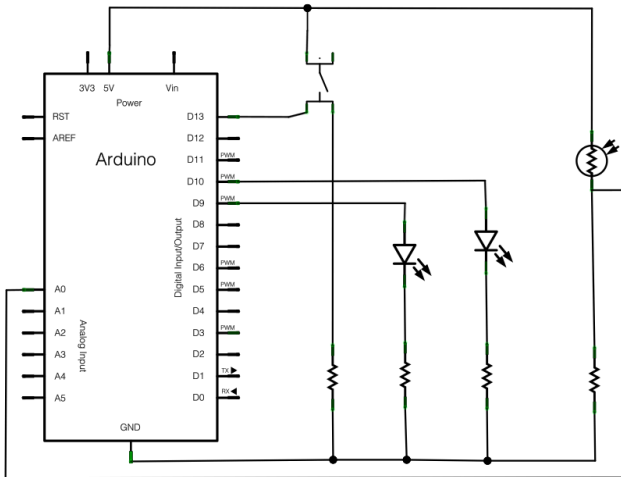
**アナログ入力**は、特定のポートの電圧を 0 から 255 の値で返すことで、各種センサの入力を検出できる。

次のプログラムでは、アナログの 0 番ポートに接続された**光センサ（CdS）**の入力値によって、画面のタートルを回転させている。

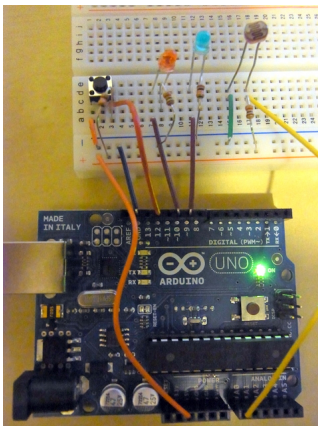
```
システム！ "arduino"使う。  
a=arduino！ 作る。  
a！（システム！ シリアルポート選択） ひらけごま。  
  
cds=a！ 0 アナログ入力。  
かめた=タートル！ 作る。  
「かめた！ ((cds！ 読む) × 10) 向き」！ 100 繰り返す。  
  
a！ とじごま。
```

## 11.7 Arduino の配線例

11.4 節から 11.6 節の動作を確認するための回路図を示す。  
抵抗値はスイッチに接続したものが 100K  $\Omega$ 、他は 10K  $\Omega$  である。



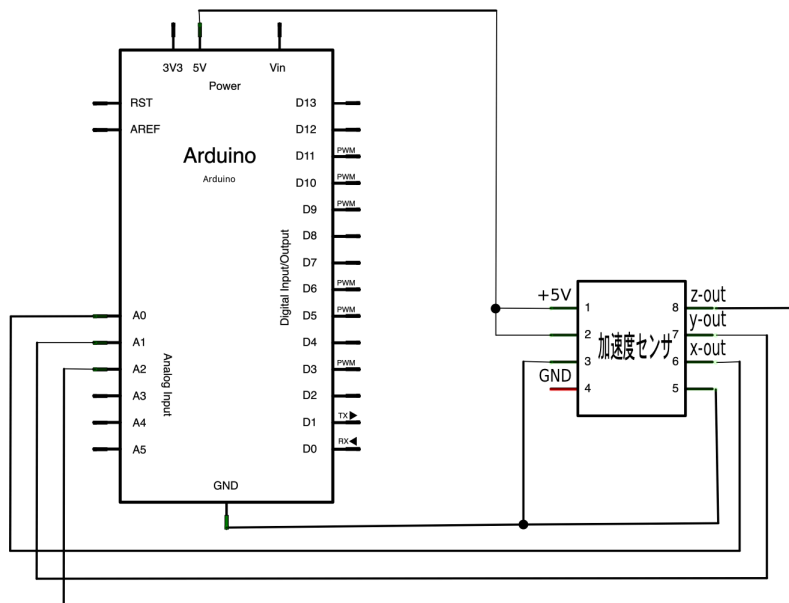
写真はブレッドボードを利用して配線したものである。配線材、LED 等の部品は、スイッチサイエンス社<sup>\*8</sup>の「Arduino をはじめようキット」のものを使用した。



<sup>\*8</sup> <http://www.switch-science.com/>

## 11.8 加速度センサの応用例

加速度センサを利用すると、傾きを検出できる。次の回路図のように、3軸加速度センサを接続することが可能である。<sup>\*9</sup>



3 軸加速度センサを利用するプログラム例を示す。ここでは 1 方向の傾きだけを利用して、Wii リモコンのように画面のタートルを操作することができる。

<sup>a9</sup> 今回は、秋月電子通商で販売されているカイオニクス社の「3 軸加速度センサモジュール KXM52-1050」で動作を確認した。



システム！ "arduino" 使う。

a=Arduino！ 作る。

a！（システム！ シリアルポート選択） ひらけごま。

かめた=タートル！ 作る。

「かめた！ 400 歩く 90 左回り」！ 4 繰り返す。

枠=かめた！ 図形を作る。枠！ -200 -200 移動する。

判定=ラベル！ 作る。

xin=a！ 0 アナログ入力。

x値=フィールド！ 作る。

「 かめた！ 5 歩く。

x=xin！ 読む。

x値！ (x) 書く。

かめた！ (x-127) 右回り。

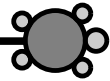
かめた：衝突=「かめた！ -10 歩く 180 右回り。判定！ "OUT" 書く 」。  
」！ 1000 繰り返す。

a！ とじろごま。



Part VII

# ロボティストを動かそう



## Activity 12

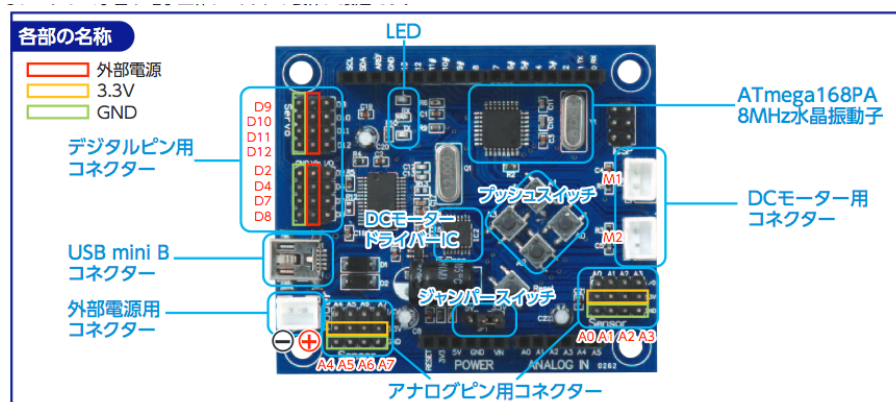


# Studuino と通信しよう

**Studuino** というデバイスを使って、LED やセンサなど外部デバイスとの入出力を試みよう。Studuino を使用するためには、ローカル版のドリトルが必要である。

## 12.1 Studuino の入手

Studuino は、株式会社アーテックが開発した汎用入出力ボードである。Arduino と互換性があり、同社のロボティストシリーズのセンサーやモーターなどと接続して使うことができる。今回は、「ロボティスト アドバンス」という製品で動作を確認した。下図は同社のサイトによる説明図である。



## 12.2 ドリトルと通信するための設定

### Windows での設定手順

以下の手順で設定する。画面に確認ダイアログが表示された場合は、すべて承認して作業を進める必要がある。

#### 1. Java をインストールする

- Web ブラウザで Java のサイト(<http://java.com/ja>)にアクセスする。
- 「無料 Java のダウンロード」からインストールする。<sup>\*1</sup>

#### 2. Arduino をインストールする

- Web ブラウザで Arduino のサイト(<http://arduino.cc>)にアクセスし、Download タブを選択する。
- ARDUINO 1.6.3 の「Windows Installer」を選択し<sup>\*2</sup>、次の画面で「JUST DOWNLOAD」を選ぶ。<sup>\*3</sup>

#### 3. ドリトルをインストールする

- ドリトルのサイト(<http://dolittle.eplang.jp>)のダウンロードから Windows 用 (dolittle237.zip) をダウンロードして任意のフォルダに展開する。
- 展開した dolittle237 フォルダをエクスプローラで開き、その中にある studuino\_setup.bat を右クリックして「管理者として実行」を選択する。<sup>\*4</sup>
- 黒いウィンドウが開いて「22 個のファイルをコピー

---

<sup>\*1</sup> Internet Explorer の「実行または保存しますか？」では「実行 (R)」を選択する。「次のプログラムにこのコンピュータへの変更を許可しますか？」のダイアログでは「はい (Y)」を選択する。

<sup>\*2</sup> 1.6.3 以外のバージョンが表示された場合は「PREVIOUS RELEASES」から 1.6.3 を探してインストールする)

<sup>\*3</sup> Internet Explorer の「実行または保存しますか？」では「実行 (R)」を選択する。「次のプログラムにこのコンピュータへの変更を許可しますか？」では「はい (Y)」を選択する。「License Agreement」では「I Agree」を選択し、次の画面で「Next」を選択する。「Destination Folder」は変更せず「Install」を選択する。「このデバイス ソフトウェアをインストールしますか？」では「インストール (I)」を選択する

<sup>\*4</sup> 「Windows によって PC が保護されました」が表示された場合は「詳細情報」から「実行」を選択する。「次のプログラムにこのコンピュータへの変更を許可しますか？」が表示された場合は「はい (Y)」を選択する

しました 続行するにはなにかキーを押してください...」と表示されたらエンターキーを押して閉じる。

#### 4. Studuino の接続

- PC の USB ポートに Studuino を接続する。
- 自動的にドライバがインストールされ、「Prolific USB-to-Serial Comm Port (COM4) デバイスドライバーソフトウェアが正しくインストールされました。」と表示される。<sup>\*5</sup>

#### 5. ドリトルの起動

- エクスプローラから dolittle.bat を実行する。<sup>\*6</sup>

## Mac での設定手順

以下の手順で設定する。画面に確認ダイアログが表示された場合は、すべて承認して作業を進める必要がある。

#### 1. Java をインストールする

- Web ブラウザで Java のサイト(<http://java.com/ja>)にアクセスする。
- 「無料 Java のダウンロード」からインストールする。

#### 2. ドリトルをインストールする

- ドリトルのサイト(<http://dolittle.eplang.jp>)のダウンロードページから Mac 用 (Dolittle237.dmg) をダウンロードする。
- ダウンロードした Dolittle237.dmg をファインダーからクリックして実行する。
- 表示された「Dolittle」を「アプリケーション」フォルダに入れる。<sup>\*7</sup>

#### 3. Studuino の接続

- USB ポートに Studuino を接続する。

---

<sup>\*5</sup> COM4 の部分は環境によって異なる場合がある。

<sup>\*6</sup> 「発行元を確認できませんでした。このソフトウェアを実行しますか？」のダイアログでは「このファイルを開く前に常に警告する (W)」のチェックを外してから「実行 (R)」を選択する。

<sup>\*7</sup> 古いバージョンが残っているというダイアログが表示された場合は「置き換える」を選択する。

#### 4. ドリトルの起動

- アプリケーションフォルダから Dolittle を実行する。

## 12.3 ドリトルからプログラムを転送する

ドリトルで記述したプログラムは次の手順で転送する。

1. ドリトルの編集画面でプログラムを記述する。
2. プログラムを実行すると「Studuino への転送を実行しますか?」と表示されるので「はい (Y)」を選択する。<sup>\*8</sup>
3. 10 秒程度待つと Studuino 基板に転送が行われ「転送完了」が表示される。「OK」を選択して閉じる。

## 12.4 センサーなどのパーツを接続する

以下に各種パーツを接続するポートを示す。<sup>\*9</sup>

- ◎と○は使用可能を、×は使用不能を表す。
- プッシュスイッチ◎を使用するとき、○の同名のポートは使用できない。たとえばプッシュスイッチ A0 を使用するとき、他のパーツは A0 を使用できない。
- DC モーター◎を使用するとき、サーボモーター○の対応するポートは使用できない。たとえば DC モーター M1 を使用するときサーボモーター D2, D4 は使用できない。<sup>\*10</sup>
- 加速度センサーは A4,A5 を同時に使用する。

<sup>\*8</sup> 「Windows セキュリティの重要な警告」が表示された場合は「アクセスを許可する (A)」を選択する。

<sup>\*9</sup> この章の例では、LED を A0 と D10 に、光センサーを A3 に、赤外線センサーを A7 に、DC モーターを M1 と M2 に接続している。

<sup>\*10</sup> V2.36 では「ブザー」「サーボモーター」には対応していない。今後のバージョンで対応される予定である。

	A0～A3	A4,A5	A6,A7	D2,D4 M1	D7,D8 M2	D9～D11	D12
加速度センサー	×	○	×	×	×	×	×
タッチセンサー	○	○	×	×	×	×	×
プッシュスイッチ	◎	×	×	×	×	×	×
それ以外のセンサー	○	○	○	×	×	×	×
LED	○	○	×	×	×	○	×
DC モーター	×	×	×	◎	◎	×	×
サーボモーター	×	×	×	○	○	○	○



## 12.5 デジタル出力による動作の確認

動作を確認するには、A0 ポートに LED を接続した状態で Studuino と PC 本体を USB ケーブルを接続し、ドリトルに次のプログラムを入力して実行する。接続した LED が 1 秒ごとに点滅すれば、プログラムの転送は正常に行われている。

システム！ "studuino" 使う。

最初に実行＝「

ST！ "A0" デジタルLED。  
」。

繰り返し実行＝「

ST！ "A0" 1 書く。  
ST！ 1000 待つ。  
ST！ "A0" 0 書く。  
ST！ 1000 待つ。  
」。

ST！ 転送。

ドリトルから Studuino を制御するプログラムは、次の形で記述する。先頭の行では、Studuino を**使う**プログラムを作成することを示している。以後、Studuino 本体を表す「ST」\*11というオブジェクトを使うことができるようになる。

システム！ "studuino" 使う。

最初に実行＝「

...  
」。

繰り返し実行＝「

...  
」。

ST！ 転送。

\*11 「スタディーノ」「studuino」「ロボ」も使用できる。

**最初に実行**の「...」の部分には、使用するセンサーの定義など、最初に 1 回だけ実行されるプログラムを書く。今回は最初に、A0 ポートを LED を接続するために初期化した。

**繰り返し実行**の「...」の部分には、何度も繰り返して実行されるプログラムを書く。今回は A0 ポートに接続した LED を 1 で光らせた後、1 秒間（1000 ミリ秒間）待ってから、LED を 0 で消灯し、1 秒間待っている。この動作を繰り返す。

**転送**を実行すると、プログラムがコンパイルされ、Studuino に転送される。

## 12.6 アナログ出力

アナログ出力では、出力する値として 0 から 255 の数値を指定できる。次のプログラムでは、最初に D10 ポートに接続された LED を「0」で消灯した後、1 秒後に「127」の明るさで点灯させ、1 秒後に「255」の明るさで点灯している。この動作を繰り返す。

システム！ "studuino" 使う。

最初に実行＝「

ST！ "D10" アナログLED。

」。

繰り返し実行＝「

ST！ "D10" 0 書く。

ST！ 1000 待つ。

ST！ "D10" 127 書く。

ST！ 1000 待つ。

ST！ "D10" 255 書く。

ST！ 1000 待つ。

」。

ST！ 転送。

## 12.7 デジタル入力

**デジタル入力**は、ポートの値を 0 と 1 で表すことで、スイッチ等の ON と OFF を検出できる。

次のプログラムでは、A1 ポートに入力があるとき (Studuino 基板上のスイッチ A1 が押されて値が 0 になったとき) に、A0 ポートの LED を 1 で点灯している。入力がないときは 0 で消灯する。

システム！ "studuino" 使う。

最初に実行＝「

ST！ "A1" スイッチ。

ST！ "A0" デジタルLED。

」。

繰り返し実行＝「

「(ST！ "A1" 読む) == 0」！ なら「

ST！ "A0" 1 書く。

」そうでなければ「

ST！ "A0" 0 書く。

」実行。

」。

ST！ 転送。

## 12.8 アナログ入力

**アナログ入力**は、特定のポートの電圧を 0 から 1023 の値で返すことで、各種センサの入力を検出できる。

次のプログラムでは、アナログの A3 ポートに接続された**光センサ**の入力値によって、LED の点灯と消灯を切り替えている。(光センサを照明に向けたり手で隠すことで明るさを変えられる。明暗を判断する値(この例では 500)は必要に応じて変更する必要がある)

```
システム！ "studuino" 使う。
```

```
最初に実行＝「
```

```
  ST！ "A3" 光センサー。
```

```
  ST！ "A0" デジタルLED。
```

```
」。
```

```
繰り返し実行＝「
```

```
  「(ST！ "A3" 読む) < 500」！ なら「
```

```
    ST！ "A0" 1 書く。
```

```
  」そうでなければ「
```

```
    ST！ "A0" 0 読む。
```

```
  」実行。
```

```
」。
```

```
ST！ 転送。
```

## 12.9 ライントレースの例

アナログ入力の実用として**ライントレースカー**を考える。ライントレースカーは左右の DC モーターを使い、地面に描かれた線のコースに沿って動く。線の上にいるかどうかを**赤外線センサー**（**赤外線フォトリフレクタ**）で判断する。

次のプログラムでは、A7 ポートに接続された赤外線センサーの入力値によって、線の上がいなければ右折し、線の上であれば左折する。（床の白黒を判断する値（この例では 500）は必要に応じて変更する必要がある）

```
システム！ "studuino" 使う。
```

```
最初に実行＝「
```

```
  ST！ DCモーター。
```

```
  ST！ "A7" 赤外線センサー。
```

```
」。
```

```
繰り返し実行＝「
```

```
  「(ST！ "A7" 読む) < 500」！ なら「
```

```
    ST！ 150 右折。
```

```
  」そうでなければ「
```

```
    ST！ 150 左折。
```

```
  」実行。
```

```
」。
```

```
ST！ 転送。
```

## 12.10 サーボモーター

**サーボモーター**を使うと、0 度から 180 の範囲で任意の角度に回転させることができる。

サーボモーターが回転している間に並行して次の命令が実行される。複数のサーボモーターを回転させる命令を続けて書くと、それらは同時に回転する。サーボモーターの回転が終るのを待ちたいときは、**待つ**などを使って適切な時間だけプログラムの実行を待つ必要がある。

次のプログラムでは、D10 ポートと D11 ポートに接続したサーボモーターの角度を 1 秒おきに変更することで回転させている。

```
システム！ "studuino" 使う。
```

```
最初に実行＝「
```

```
    ST！ "D10" サーボモーター。
```

```
    ST！ "D11" サーボモーター。
```

```
」。
```

```
繰り返し実行＝「
```

```
    ST！ "D10" 180 書く。
```

```
    ST！ "D11" 0 書く。
```

```
    ST！ 1000 待つ。
```

```
    ST！ "D10" 90 書く。
```

```
    ST！ "D11" 90 書く。
```

```
    ST！ 1000 待つ。
```

```
」。
```

```
ST！ 転送。
```





Part VIII

# MYU ロボを動かそう



## Activity 13



# ロボットの準備

ロボットなど外部機器制御を行うための環境設定と使う教材を解説する。

## 13.1 ドリトルとの接続

ロボットを制御するためには、ローカル版のドリトルが必要である。ドリトルが動いているコンピュータとロボットは、シリアルケーブルで接続する。コンピュータにシリアルポートが存在しない場合には、市販の USB シリアル変換ケーブルを利用する。<sup>\*1</sup>ドリトルから通信するポートは、システムオブジェクトの**シリアルポート選択**で実行時に選択できる。<sup>\*2</sup>

## 13.2 ロボット

本書では **MYU ロボ** というロボットでのプログラミングを解説する。

本書で扱うロボットは、スタジオミュウで開発・販売されている製品である。この製品は、CPU に PIC を採用している。この CPU にはメモリや I/O（外部インターフェース）の機能が搭載されており、ドリトルから転送したプログラムを解釈し

<sup>\*1</sup> 本書では、Windows は秋月電子通商の「USB・シリアル変換ケーブル」で、Macintosh ではパッファローコクヨサプライの「Arvel USB シリアルケーブル SRC06USB」にチップメーカーのサイト (<http://www.ftdichip.com/Drivers/VCP.htm>) からドライバをインストールして動作を確認しています。

<sup>\*2</sup> あらかじめ調べる場合は、Windows であればデバイスマネージャの「ポート (COM と LPT)」の COM の部分を調べるができる。Macintosh では設定が必要な場合がある。「はじめに」(p.iv) を参照。



システム！ "myurobo" 使う。

ロボ太＝ミュウロボ！ 作る。

ロボ太！ひらけごま。

前進ボタン＝ボタン！ "前進" 作る -220 150 位置。

前進ボタン：動作＝「ロボ太！前進」。

後進ボタン＝ボタン！ "後進" 作る 0 150 位置。

後進ボタン：動作＝「ロボ太！後退」。

左前ボタン＝ボタン！ "左回り" 作る -220 100 位置。

左前ボタン：動作＝「ロボ太！左回り」。

右前ボタン＝ボタン！ "右回り" 作る 0 100 位置。

右前ボタン：動作＝「ロボ太！右回り」。

停止ボタン＝ボタン！ "停止" 作る -110 40 位置。

停止ボタン：動作＝「ロボ太！停止」。

音1ボタン＝ボタン！ "音1" 作る -220 -20 位置。

音1ボタン：動作＝「ロボ太！15 電子音」。

音2ボタン＝ボタン！ "音2" 作る -220 -70 位置。

音2ボタン：動作＝「ロボ太！16 電子音」。

ロボ太！とじろごま。

## Activity 14



# ロボットを動かそう

## 14.1 プログラムの記述

MYU ロボットの制御プログラムの基本形を示す。先頭の行では、MYU ロボットを**使う**プログラムを作成することを示している。2 行目では、ドリトル上でロボットを扱うためのロボ太という名前の **MYU オブジェクト**を作っている。3 行目からは、ロボットに転送するプログラムをメソッドとして定義している。ロボットに転送するプログラムは、**はじめロボットとおわりロボット**の間に記述する。

```
システム！ "myurobo" 使う。
ロボ太＝ミュウロボ！ 作る。
実行命令＝「ロボ太！ はじめロボット。
```

```
//ここに制御プログラムを書く
```

```
ロボ太！ おわりロボット」。
```

実際のプログラムを示す。ロボット制御のプログラムでは、「前進」や「右回り」の単位は 0.1 秒である。次のプログラムは、ロボットが 1 秒間前進し、0.5 秒間右に回転する動作を表している。表 14.1 に使用できる命令の例を示す。

```
システム！ "myurobo" 使う。
ロボ太＝ミュウロボ！ 作る。
実行命令＝「ロボ太！ はじめロボット。
    ロボ太！ 10 前進 5 右回り。
ロボ太！ おわりロボット」。
```

表 14.1 ロボットの命令（一部）

命令	用途	使用例
はじめロボット	制御プログラムの先頭を示す	ロボ太！ はじめロボット。
おわりロボット	制御プログラムの末尾を示す	ロボ太！ おわりロボット。
前進	両輪の前転を開始し、パラメータの値×0.1 秒間だけ待つ	ロボ太！ 10 前進。
後退	両輪の後転を開始し、パラメータの値×0.1 秒間だけ待つ	ロボ太！ 10 後退。
右回り	左車輪の前転、右車輪の後転を開始し、パラメータの値×0.1 秒間だけ待つ	ロボ太！ 10 右回り。
左回り	右車輪の前転、左車輪の後転を開始し、パラメータの値×0.1 秒間だけ待つ	ロボ太！ 10 右回り。
停止	すべてのモータを停止させた後、パラメータの値×0.1 秒間だけ待つ	ロボ太！ 10 停止。
繰り返し脱出	繰り返しブロックの中に書き、繰り返しを中断して次に進む	ロボ太！ 繰り返し脱出。
繰り返す	指定した回数だけブロックを実行する	「ロボ太！ 5 ブザー 2 時間」！ 3 繰り返す。
入力あり	指定した番号の入力が ON ならブロックを実行する	「ロボ太！ 2 入力あり」！ なら「ロボ太！ 10 後退」実行。

14.2 プログラムの実行

プログラムを作成したら、ロボットに転送して実行する。  
ドリトルを実行しているコンピュータとロボットは、あらかじめ転送ケーブルで接続しておく。  
ドリトルの「実行！」ボタンを押すと、実行画面に「画面実行」ボタンと「転送」ボタンが表示される。



画面実行

転送

「画面実行」ボタンを押すと、画面上のミュウロボがプログラムを実行する。エラーが表示された場合は、プログラムを修正してから再度実行する。

「転送」ボタンを押すと、ロボットへの転送が行われる。「ピ・・」という音が鳴り、最後に「ピピ」という音が鳴れば、正常に転送が終了したことが分かる。正常に転送されたら、転送ケーブルをロボットから抜く。

ロボットに転送されたプログラムの実行を開始する方法は2種類ある。ひとつは、先頭のタッチスイッチを押す方法である。今回のテキストでは、この方法で説明している。もうひとつの方法としては、「！ はじめロボット」に続けて**パワーオンスタート**を書いておくと、電源を入れたときにプログラムが実行される。これらの方法は、好みや状況に応じて使い分けることができる。

転送されたプログラムは電源を切った後もロボットに記憶されている。電源を入れ、タッチスイッチを押すことで再度実行することが可能である。

## 14.3 モーターの性質

このロボットでは、「前進」などのモーターの動作に関する命令が実行されると、次にモーターに関する命令が実行されるまで動作を継続する性質がある。そのため、確実に1秒間だけモーターを動作させたい場合には、「10 前進」と書くだけでなく、「10 前進 停止」のように、明示的に動作の停止を記述する必要がある。

ロボットがこのような性質を持つことで、モーターの動作と並行して「スイッチの入力を監視する」「音を鳴らす」といった処理を行うことが可能になる。

## 14.4 繰り返し

**繰り返し**を使うと、命令を繰り返して実行できる。次のプログラムは、「前進して右回り」を10回繰り返す。結果としてロボットは多角形を描くように移動する。

システム！ "myurobo" 使う。

ロボ太=ミュウロボ！ 作る。

実行命令＝「ロボ太！ はじめロボット。

「ロボ太！ 10 前進 5 右回り」！ 10 繰り返す。

ロボ太！ おわりロボット」。



繰り返しの回数を省略した場合は、無限回という意味になり、**無限ループ**として、いつまでも繰り返して命令を実行し続ける。無限ループを使う場合には、ある条件が成り立ったときに繰り返しから抜けるために**繰り返し脱出**と組み合わせて使うことが多い。

次のプログラムを実行すると、ロボットは多角形を描くように移動するが、タッチスイッチで障害物を検出するとループを抜ける。

システム！ "myurobo" 使う。

ロボ太=ミュウロボ！ 作る。

実行命令＝「ロボ太！ はじめロボット。

「ロボ太！ 10 前進。

「ロボ太！ 2 入力あり」！ なら「ロボ太！ 繰り返し脱出」実行。

ロボ太！ 5 右回り。

」！ 繰り返す。

ロボ太！ 10 後退 停止。

ロボ太！ おわりロボット」。





ロボットを制御するプログラムは画面のオブジェクトを操作するタートルグラフィックスと似ているが、前進や回転を距離や角度で指定するのではなく、モータを回転する時間で指定する点が異なっている。実際にどれくらいの距離を移動し、どれくらいの角度を回転するのかは、実行して確認する必要がある。

ゴールに到達できるように何度も試行錯誤を行っている、最初に調整した動きとずれてくることがある。プログラムで指定するのはモータを回す時間だけであるため、摩擦や電池の消耗などの原因により、実行を繰り返すうちに、同じプログラムでも動作が微妙に異なる場合がある。つまりこの方法では、ある状態のときにうまくゴールに到達できたとしても、次に実行したときにゴールに到達できる保証はない。

## 15.3 スイッチ入力検出

今回使用しているロボットの先頭には衝突検出用の触覚スイッチが取り付けられており、右が2番、左が3番の入力に接続されている。このスイッチを利用すると、前進して壁に衝突したことを検知できるので、電池の消耗や個体差に影響されない確実な動作を実現できる。

ここでは、「前進し、障害物を感知したら後退して向きを変える」という動作を繰り返すプログラムを説明する。

次のプログラムで、

「ロボ太！ 2 入力なし」！の間「ロボ太！ 前進」実行。

により1番のスイッチが押されていない間**ブロック**(「...」)を実行する。その結果、スイッチが押されるまで「前進」が繰り返し実行され、スイッチが押されたときに

ロボ太！ 10 後退 10 右回り。

が実行されることになる。表 15.1 に命令の例を示す。

システム！ "myurobo" 使う。  
 ロボ太＝ミュウロボ！ 作る。  
 実行命令＝「ロボ太！ はじめロボット。  
     ロボ太！ 10 前進。  
     「  
         「ロボ太！ 3 入力なし」！ の間「ロボ太！ 前進」  
 実行。  
     ロボ太！ 10 後退 10 右回り。  
     」！ 繰り返す。  
 ロボ太！ おわりロボット」。

表 15.1 ロボットの命令（一部）

命令	用途	使用例
<b>停止</b>	すべてのモータを停止させた後、パラメータの値（× 0.1 秒間）だけ待つ	ロボ太！ 10 停止。
<b>リミットスイッチ</b>	1 番センサに入力があるまで直前の動作を続ける	ロボ太！ リミットスイッチ。
<b>電子音</b>	第 1 パラメータの時間だけ、第 2 パラメータの音程でブザーを鳴らす。音程は数字が大きいほど低い	ロボ太！ 10 40 ブザー。
<b>繰り返す</b>	指定した回数だけブロックを実行する	「ロボ太！ 5 ブザー 2 時間」！ 3 繰り返す。
<b>繰り返し脱出</b>	繰り返しブロックの中に書き、繰り返しを中断して次に進む	ロボ太！ 繰り返し脱出。
<b>入力あり... 実行</b>	指定した番号の入力が ON ならブロックを実行する	「ロボ太！ 2 入力あり」！ なら「ロボ太！ 10 後退」実行。
<b>の間... 実行</b>	指定した条件が成り立つ間、ブロックを繰り返して実行する	「ロボ太！ 2 入力あり」！ の間「ロボ太！ 後退」実行。
<b>実行脱出</b>	「の間... 実行」ブロックの中に書き、繰り返しを中断して次に進む	ロボ太！ 実行脱出。

## 参考 1: 入力があるときだけ実行する方法

動きながらスイッチの入力を検出するプログラムは、複数の方法で書くことができる。ここでは繰り返しの中で、入力が

あったときだけ特定の動作を行う方法を示す。書き方は異なるが、前の例と動作は同じである。

次のプログラムで、**入力あり**は3番のスイッチが押されているときだけ**ブロック**(「...」)を実行する。その結果、スイッチが押されるまで「前進」が繰り返し実行され、スイッチが押されたときに

ロボ太！ 10 後退 10 右回り。

が実行されることになる。

システム！ "myurobo" 使う。

ロボ太＝ミュウロボ！ 作る。

実行命令＝「ロボ太！ はじめロボット。

ロボ太！ 10 前進。

「ロボ太！ 前進。

「ロボ太！ 3 入力あり」！ なら「ロボ太！ 10 後退 10 右回り」実行。

」！ 繰り返す。

ロボ太！ おわりロボット」。

## 参考 2: 入力があるまでプログラムの実行を止める方法

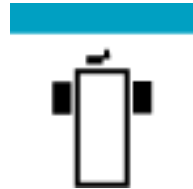
ここでは繰り返しの中で、入力がない間、特定の動作を行う方法を示す。書き方は異なるが、前の例と動作は同じである。

次のプログラムで、**リミットスイッチ**は1番入力に接続されたセンサ（スイッチ）が押されるまでプログラムの進行を止める。その結果、ロボットは直前の「前進」の実行を維持することになり、前進を続ける。スイッチが押されたときには、プログラムの進行が再開されるため、その後の

ロボ太！ 10 後退 10 右回り。

が実行されることになる。

システム！ "myurobo" 使う。  
 ロボ太＝ミュウロボ！ 作る。  
 実行命令＝「ロボ太！ はじめロボット。  
     ロボ太！ 10 前進。  
     「ロボ太！ 前進。  
     ロボ太！ リミットスイッチ。  
     ロボ太！ 10 後退 10 右回り。  
     」！ 繰り返す。  
 ロボ太！ おわりロボット」。



## 15.4 ユーザー定義命令

**ユーザー定義命令**は独自の命令を定義する。

ロボットオブジェクト：命令＝「...」。

次のプログラムでは、音楽の演奏と、障害物を感知したときの動作を「演奏」と「Ｕターン」という命令として定義している。実行する命令は

「ロボ太！ 2 入力あり」！ なら「ロボ太！ 演奏 Ｕターン」実行。

と記述できるため、「2 番の入力があった場合はメロディを演奏してＵターンする」のように、プログラムを素直に読み下せるようになり、プログラムの読みやすさを向上させることができる。

システム！ "myurobo" 使う。

ロボ太＝ミュウロボ！ 作る。

ミュウロボ：演奏＝「ロボ太！

2 48 電子音 2 46 電子音 2 44 電子音 2 42 電子音

2 40 電子音 2 38 電子音 2 36 電子音 2 34 電子音

」。

ミュウロボ：Uターン＝「ロボ太！ 10 後退 10 右回り」。

実行命令＝「ロボ太！ はじめロボット。

ロボ太！ 演奏。

「ロボ太！ 前進。

「ロボ太！ 2 入力あり」！ なら「ロボ太！ 停止演奏 Uターン」実行。

」！ 繰り返す。

ロボ太！ おわりロボット」。

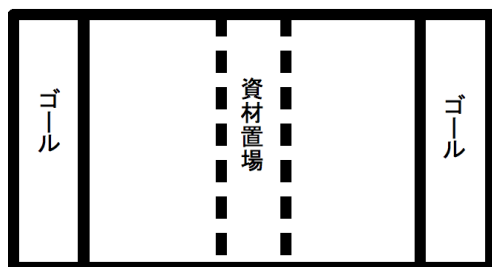
## Activity 16



# ものを運ぼう

## 16.1 想定するコート

この Activity では、ロボットにアームとモータを追加することで、物を運んだり仕事をさせるプログラムを考える\*<sup>1</sup>。想定するコートを図に示す。大きさは 180cm × 90cm（たたみ一畳分）のコンパネ（建築工事用の合板）を想定している。コートの中央に 2 × 4 材を置き、その上に荷物としてスチロールなどの緩衝材をばらまいておく。

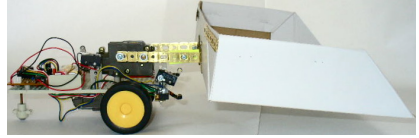
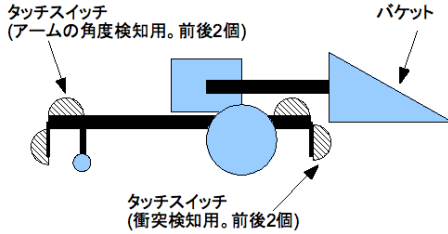


## 16.2 ロボットの拡張

ロボットには 3 個目のモータと、荷物を運ぶためのバケットとそれを支えるアームを取り付ける。アームの回転を検知するために、本体の上部にタッチスイッチを 2 個取り付け、衝突検知用のタッチスイッチも前後に取り付けて配線する。ロボットの構造を図に示す。ここでは、前の衝突検知用スイッチを 1 番、前の角度検知用スイッチを 2 番、後ろの角度検知用スイ

\*<sup>1</sup> この Activity には、「3 軸ミュウロボ基本セット」（13.2 節）が必要です。

チを3番、後ろの衝突検知用スイッチを4番にした。



## 16.3 ものを運ぶプログラム

中央の資材置場からゴールまで荷物を運ぶプログラムの流れは次のようになる。

1. ぶつかるまで前進: 資材置場に向かって進み、前のタッチスイッチが押されたら止まる。
2. アームを持ち上げる: アームを少し持ち上げて、バケットの中に資材を入れる。
3. ぶつかるまで後退: ゴールに向かって後退し、後ろのタッチスイッチが押されたら止まる。
4. 荷物を後ろに落とす: アームを後ろに回し、後ろにある角度検知用のタッチスイッチが押されたら止まる。
5. アームを戻す: アームを前に回し、前にある角度検知用のタッチスイッチが押されたら止まる。

この流れをプログラムで書くと次のようになる。

システム! "myurobo" 使う。

ロボ太=ミュウロボ! 作る。

実行命令=「ロボ太! はじめロボット。

ロボ太! パワーオンスタート。

「ロボ太! 1 入力なし!」の間「ロボ太! 前進」実行。

ロボ太! 5 モーター左。

「ロボ太! 4 入力なし!」の間「ロボ太! 後退」実行。

「ロボ太! 3 入力なし!」の間「ロボ太! モーター左」実行。

「ロボ太! 2 入力なし!」の間「ロボ太! モーター右」実行。

ロボ太! おわりロボット」。



## 16.4 命令の定義

プログラムが複雑になってくると、どの行でどのような処理をしているのかが、分かりづらくなってくる。このようなときは**ユーザー定義命令**を使い、プログラムの処理に名前を付けることで理解しやすいプログラムにできる。

この流れをプログラムで書くと次のようになる。それぞれの処理に「ぶつかるまで前進」のような名前を付けてユーザー定義命令にしている。プログラムの最初でこのような定義を行っておくことで、プログラムの中で個々の処理を具体的なイメージを持ちながら読むことが可能になる。

システム！ "myurobo" 使う。

ロボ太＝ミュウロボ！ 作る。

ミュウロボ：ぶつかるまで前進＝「ロボ太！ 1 入力なし」！の間「ロボ太！ 前進」実行」。

ミュウロボ：アームを持ち上げる＝「ロボ太！ 5 モーター左」。

ミュウロボ：ぶつかるまで後退＝「ロボ太！ 4 入力なし」！の間「ロボ太！ 後退」実行」。

ミュウロボ：荷物を後ろに落とす＝「ロボ太！ 3 入力なし」！の間「ロボ太！ モーター左」実行」。

ミュウロボ：アームを戻す＝「ロボ太！ 2 入力なし」！の間「ロボ太！ モーター右」実行」。

実行命令＝「ロボ太！ はじめロボット。

ロボ太！ パワーオンスタート。

ロボ太！ ぶつかるまで前進。

ロボ太！ アームを持ち上げる。

ロボ太！ ぶつかるまで後退。

ロボ太！ 荷物を後ろに落とす。

ロボ太！ アームを戻す。

ロボ太！ おわりロボット」。

表 16.1 ロボットの命令

命令	用途	使用例
はじめロボット	制御プログラムの先頭を示す	ロボ太！ はじめロボット。
おわりロボット	制御プログラムの末尾を示す	ロボ太！ おわりロボット。
パワーオンスタート	電源オンで実行を開始する	ロボ太！ パワーオンスタート。
前進	両輪の前転を開始し、パラメータの値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 前進。
後退	両輪の後転を開始し、パラメータの値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 後退。
右回り	左車輪の前転、右車輪の後転を開始し、 パラメータの値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 右回り。
左回り	右車輪の前転、左車輪の後転を開始し、 パラメータの値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 右回り。
停止	すべてのモータを停止させた後、パラ メータの値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 停止。
右前	右車輪の前転を開始し、パラメータの 値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 右前。
左前	左車輪の前転を開始し、パラメータの 値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 左前。
右後	右車輪の後転を開始し、パラメータの 値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 右後。
左後	左車輪の後転を開始し、パラメータの 値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 左後。
モーター右	第 3 モーターを右回転し、パラメータ の値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 モーター右。
モーター左	第 3 モーターを左回転し、パラメータ の値 $\times 0.1$ 秒間だけ待つ	ロボ太！ 10 モーター左。
繰り返す	指定した回数だけブロックを実行する	「ロボ太！ 5 ブザー 2 時間」！ 3 繰り返す。
繰り返し脱出	繰り返しブロックの中に書き、繰り返 しを中断して次に進む	ロボ太！ 繰り返し脱出。
入力あり	指定した番号の入力が ON ならブロッ クを実行する	「ロボ太！ 2 入力あり」！ なら「ロ ボ太！ 10 後退」実行。
入力なし	指定した番号の入力が OFF ならブロッ クを実行する	「ロボ太！ 2 入力なし」！ なら「ロ ボ太！ 10 後退」実行。
リミットスイッチ	1 番センサに入力があるまで直前の動 作を続ける	ロボ太！ リミットスイッチ。
の間... 実行	指定した条件が成り立つ間、ブロック を繰り返して実行する	「ロボ太！ 2 入力あり」！ の間「ロ ボ太！ 後退」実行。
実行脱出	「の間... 実行」ブロックの中に書き、 繰り返しを中断して次に進む	ロボ太！ 実行脱出。

表 16.2 ロボットの命令（続き）

命令	用途	使用例
ブザー	パラメータの回数だけブザーを鳴らす	ロボ太！ 10 ブザー。
電子音	第 1 パラメータの時間だけ、第 2 パラメータの音程でブザーを鳴らす。音程は数字が大きいほど低い	ロボ太！ 10 40 ブザー。



Part IX

# ネットワークで通信 しよう



## Activity 17



# ネットワーク通信

ドリトルではサーバーを起動することで、ネットワークに接続された他のコンピュータ上のドリトルと通信することが可能になる。

## 17.1 サーバーの起動

サーバーはドリトルの編集画面から「server」のチェックボックスをクリックすることで起動することができる。警告のウィンドウが表示された場合には、「ブロックを解除する」を選択する。

起動すると、もうひとつの画面が表示される。ドリトルの実行画面と似ているが、画面下のボタンは2つしかない。これがサーバーのウィンドウである。



## 17.2 IP アドレスの確認

ドリトルがサーバーと通信するときは、サーバーが起動しているコンピュータを指定する必要がある。ドリトルがサーバーと同じコンピュータで動作している場合は、コンピュータ名として **localhost** を指定することで通信できる。

ドリトルがサーバーと異なるコンピュータで動作している場合は、サーバーの動作している **コンピュータ名** または **IP アドレス** を指定する。ドリトルが動作しているコンピュータの IP アドレスは、サーバー画面の下部に表示されている。（「192.168.11.8」の部分はコンピュータによって異なる）



## 17.3 サーバーとの接続

サーバーと通信するプログラムでは、最初にサーバーとの**接続**を行う。次のプログラムはドリトルが動いているコンピュータで動作しているサーバーと接続する。異なるコンピュータで動作している場合には、「localhost」の代りにサーバーが動作しているコンピュータを名前か IP アドレスで指定する。

```
サーバー！ "localhost" 接続。
```

## 17.4 オブジェクトの書き込み

あるコンピュータ上で作ったタートルやボタンなどの各種のオブジェクトをサーバーに登録することで、他のコンピュータに受け渡すことができる。

サーバーオブジェクトの**書く**という命令により、指定したオブジェクトの複製をサーバー上に名前を付けて登録できる。次のプログラムでは、「カメ太」という名前のタートルオブジェクトを、サーバーに「kameta」という名前で登録している。

```
サーバー！ "localhost" 接続。  
カメ太=タートル！ 作る。  
サーバー！ "kameta"   (カメ太) 書く。
```

## 17.5 オブジェクトの読み出し

サーバーに書き込んだオブジェクトは、**読む**という命令により、読み出して利用することができる。次のプログラムでは、サーバー上に「kameta」という名前で登録されたタートルオブジェクトを、使用中のドリトル上に複製し「カメ吉」という名前を付けている。



サーバー！ "localhost" 接続。  
カメ吉=サーバー！ "kameta" 読む。  
カメ吉！ 100 歩く。




## Activity 18



# チャットを作ろう

ネットワーク機能を使って、教室の中で**チャット**（画面でのおしゃべり）をするプログラムを作ってみよう。

## 18.1 メッセージを送信する

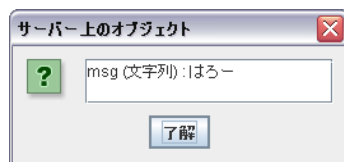
最初に、メッセージを送るプログラムを作る。画面にメッセージを入力する入力欄を作り、そこにメッセージを入力して**リターンキー**（Enter キー、)を押すと、メッセージがサーバーに書き込まれるようにする。

次のプログラムでは、「送信フィールド」というフィールドオブジェクトを作り、リターンキーを押したときに、送信フィールドに書かれた文字列をサーバーに「msg」という名前で書き込む。送ったことが分かるように、最後にフィールドの中身を「クリア」で空にするようにした。

```
サーバー！ "localhost" 接続。
送信フィールド=フィールド！ 作る。
フィールド：動作＝「
    サーバー！ "msg"  （送信フィールド！ 読む）書く。
    送信フィールド！ クリア。
」。
```

はろー

サーバーに接続するプログラムを実行すると、ドリトルの編集画面には「サーバー」というボタンが表示される。このボタンを押すことで、サーバーに書き込まれたオブジェクトを確認できる。「msg（文字列）：はろー」から、「はろー」という文字列が msg という名前で書き込まれていることが分かる。



## 18.2 メッセージを受信する

次のプログラムは、メッセージを受信するプログラムである。実行すると、サーバーから「msg」という文字列オブジェクトを読み出し、ラベルに表示する。

サーバー！ "localhost" 接続。  
受信メッセージ=サーバー！ "msg" 読む。  
ラベル！（受信メッセージ）作る。

はろー

送信と受信のプログラムを1台のコンピュータで実行する場合には、次のようにする。

- 1台のコンピュータでドリトルを2つ起動する。
- それぞれの画面で送信と受信のプログラムを入力する。
- どちらか一方でサーバーを起動する。
- 送信側のプログラムを実行し、メッセージを書き込んだ後、受信側のプログラムでメッセージを受信する。

ネットワークで接続された複数台のコンピュータで動作を確認する場合には、次のようにする。

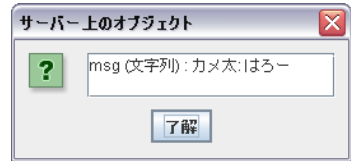
- 複数台のコンピュータでドリトルを起動する。2台以上であれば何台でも構わない。
- 1台のドリトルでサーバーを起動する。（他のドリトルでは、サーバーを「終了」ボタンで閉じておくとよい）
- ドリトルで送信または受信のプログラムを入力する。
- プログラムの「localhost」の部分にサーバーを実行しているコンピュータのIPアドレスを記述する。
- 送信側のプログラムを実行し、メッセージを書き込んだ後、受信側のプログラムでメッセージを受信する。<sup>\*1</sup>

<sup>\*1</sup> メッセージを複数のドリトルから送信することもできる。新しいメッセージが書き込まれると前のメッセージは上書きされてしまうので、複数人で実行する場合には、「書いたよ」と声を掛け合いながら送信と受信のプログラムを実行するとよい。

## 18.3 発言に名前を入れる

複数の人で発言を書き込む場合には、誰が書いた発言かが分かると便利である。そこで、「はろー」という発言するときには、前に自分の名前を入れて、「カメ太: はろー」という形で書き込むことを考える。

次のプログラムでは、「名前」という変数に名前を入れている。ここでは「カメ太」としているが、実際にはそれぞれの自分の名前を書いておく。サーバーには名前に ":" と発言の文字列を連結してから書き込んでいる。「連結」は、文字列に他の文字列を連結する命令である。



```
名前="カメ太"。
サーバー! "localhost" 接続。
送信フィールド=フィールド! 作る。
フィールド:動作=「
  サーバー! "msg" (名前! ":" " (送信フィールド! 読む) 連結) 書く。
  送信フィールド! クリア。
」。
```

## 18.4 受信の自動化 (1)

他の人の発言はいつ書き込まれるか分からないので、受信プログラムを何度も実行して新しい発言をチェックするのは大変である。そこで、定期的にサーバーと通信して、新しい発言を表示するプログラムを考えてみる。

次のプログラムを実行すると、サーバーから 1 秒間隔でメッセージを読み、「受信表示」というラベルに出力する。この動作を 600 回 (10 分間) 繰り返す。

```
サーバー! "localhost" 接続。
受信表示=ラベル! 作る 300 45 大きさ。
タイマー! 作る 1秒 間隔 600 回数「
  受信表示! (サーバー! "msg" 読む) 書く。
」実行。
```

カメ太: はろー

## 18.5 受信の自動化 (2)

前のプログラムでは、受信したメッセージは1秒ごとに自動的に書き換えられるため、よく見ていないと読まないうちに次のメッセージに進んでしまう。そこで、やり取りしたメッセージを画面に残すプログラムを考える。次のプログラムでは、リストオブジェクトで複数個のメッセージを表示している。

```
サーバー! "localhost" 接続。
受信表示=リスト! 作る 300 400 大きさ。
タイマー! 作る 1秒 間隔 600 回数「
    受信メッセージ=サーバー! "msg" 読む。
    受信表示! (受信メッセージ) 書く。
」実行。
```

```
カメ太: はろー
カメ太: はろー
カメ太: はろー
カメ太: はろー
カメ太: こんにちは
カメ太: こんにちは
```

実行すると、1秒ごとにサーバーと通信し、受信したメッセージを追記する。

実行結果を見ると分かるように、このままでは同じメッセージが何度も書かれてしまう。そこで、新しい発言であることをチェックするために、直前の発言と比較し、異なっている場合だけリストに書くようにプログラムを修正する。

次のプログラムでは、サーバーからメッセージを読むたびに、受信したメッセージ(「受信メッセージ」)を直前のメッセージ(「直前メッセージ」)と比較し、異なる場合だけ表示している。そして比較した後で、受信したメッセージを直前のメッセージに代入している。

```
カメ太: はろー
カメ太: こんにちは
```

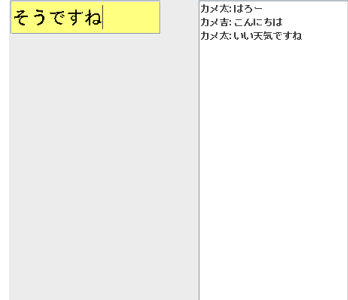
```
サーバー! "localhost" 接続。
受信表示=リスト! 作る 200 400 大きさ 0 200 位置。
直前メッセージ=""。
タイマー! 作る 1 間隔 600 回数「
    受信メッセージ=サーバー! "msg" 読む。
    「受信メッセージ!=直前メッセージ」! なら「受信表示! (受信メッセージ) 書く」実行。
    直前メッセージ=受信メッセージ。
」実行。
```

実行すると、1 秒ごとにサーバーと通信し、新しいメッセージがあったときだけ、画面に表示する。

## 18.6 チャットプログラム

ここまで作ってきた送信プログラムと受信プログラムを組み合わせると、送信と受信の両方を扱うチャットプログラムを作ることができる。

次のプログラムは、18.3 節と 18.5 節で作った 2 つのプログラムを結合させて作ったプログラムである。送信するメッセージを入力する「メッセージ」フィールドと、受信したメッセージを表示する「受信メッセージ」リストが重ならないように位置を調整している。



サーバー！ "localhost" 接続。

サーバー！ "msg" "" 書く。

名前="カメ太"。

メッセージ=フィールド！ 作る 200 45 大きさ -250 200 位置。

フィールド：動作＝「

サーバー！ "msg" (名前！ ": " (メッセージ！ 読む) 連結) 書く。

メッセージ！ クリア。

」。

受信表示=リスト！ 作る 200 400 大きさ 0 200 位置。

直前メッセージ=""。

タイマー！ 作る 1 間隔 600 回数「

受信メッセージ=サーバー！ "msg" 読む。

「受信メッセージ!=直前メッセージ」！ なら「受信表示！（受信メッセージ）書く」実行。

直前メッセージ=受信メッセージ。

」実行。

実行すると、送信するメッセージの入力欄と受信したメッセージの表示欄が画面に表示される。これは「カメ吉」がメッセージを送ろうとしている画面である。


# Activity 19



## 音楽を交換しよう

ネットワーク機能を使って、教室の中で友だちと音楽データを交換するプログラムを作ってみよう。

### 19.1 音楽を演奏する

Activity 8 で扱った音楽演奏を思い出してみよう。ドリトルでは、「ドレミー」のようにメロディを書いて音楽を演奏できる。次のプログラムでは、メロディを入力する「曲」というフィールドオブジェクトを作り、**リターンキー**（Enter キー、)を押したときにメロディオブジェクトにフィールドに書かれた旋律を追加して演奏する。

どれみー

曲=フィールド！ 作る 600 45 大きさ -250 100 位置。  
曲：動作=「メロディ！ 作る（自分！ 読む）追加 演奏」。

### 19.2 音楽を送信する


入力した音楽をサーバーに書き込めるようにする。次のプログラムでは、「送信」ボタンを押したときに、曲名とメロディをサーバーに書き込む。

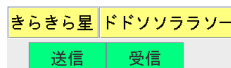
きらきら星	ドレミファミレドー
送信	

サーバー！ "localhost" 接続。  
曲=フィールド！ 作る 600 45 大きさ -250 100 位置。  
曲：動作=「メロディ！ 作る（自分！ 読む）追加 演奏」。  
曲名=フィールド！ 作る 130 45 大きさ -380 100 位置。  
送信ボタン=ボタン！ "送信" 作る -350 50 位置。  
送信ボタン：動作=「サーバー！（曲名！ 読む）（曲！ 読む）書く」。

実行すると、サーバーにメロディの文字列が書き込まれる。

## 19.3 音楽を受信できるようにする

サーバーから音楽を受信して演奏できるようにする。次のプログラムでは、曲名を入力して「受信」ボタンを押したときに、サーバーからその曲のメロディをダウンロードして「曲」フィールドに表示する。曲フィールドでリターンキー（Enter キー、)を押すことで、ダウンロードした曲が演奏される。このように、ひとつのフィールドを送信と受信のために共通に使うこともできる。




サーバー！ "localhost" 接続。  
 曲=フィールド！ 作る 600 45 大きさ -250 100 位置。  
 曲：動作＝「メロディ！ 作る （自分！ 読む）追加 演奏」。  
 曲名=フィールド！ 作る 130 45 大きさ -380 100 位置。  
 送信ボタン=ボタン！ "送信" 作る -380 50 位置。  
 送信ボタン：動作＝「サーバー！（曲名！ 読む）（曲！ 読む）書く」。  
 受信ボタン=ボタン！ "受信" 作る -230 50 位置。  
 受信ボタン：動作＝「曲！（サーバー！（曲名！ 読む）読む）書く」。

ネットワークで接続された複数台のコンピュータで動作を確認する場合には、次のようにする。

- 複数台のコンピュータでドリトルを起動する。2 台以上であれば何台でも構わない。
- それぞれのドリトルに上のプログラムを入力する。
- どれか 1 台のドリトルでサーバーを起動する。他のドリトルでは、サーバーを「終了」ボタンで閉じておくとよい。
- プログラムの「localhost」の部分にサーバーを実行しているコンピュータ名を記述する。コンピュータ名が分からない場合は、サーバーの IP アドレスを 17.2 節の手順で調べて記述する。
- 曲を入力し、分かりやすい名前を付けてサーバーに書き込む。
- ドリトルの編集画面から「サーバー」ボタンを押してサーバー上のオブジェクトを確認する。自分の書き込んだ曲や、他の人が書き込んだ曲を一覧できる。確認したら「了

解」ボタンでダイアログを閉じる。

- プログラムの実行画面で、他の人の曲名を入力してから「受信」ボタンを押して受信し、表示された曲のフィールドでリターンキー（Enter キー、)を押して、ダウンロードした曲を演奏する。これは音楽や映像をインターネットのサーバーに書き込み、それをいろいろな人が視聴することに相当する。
- なお、受信時に曲が見つからないときは、メロディの欄に「値が存在しない」という意味の**未定義**オブジェクトである「[undef]」が表示される。

## 19.4 長い曲を交換（ダウンロードとストリーミング）

携帯電話やパソコンでは、インターネットからダウンロードした音楽を聴くことができる。また、ダウンロードせずにストリーミングとして聴いていくこともできる。ここでは、ドリトルで作った音楽を、ダウンロードとストリーミングという2種類の方法で聴くためのプログラムを作成する。授業で学習する場合には、先生など1人が送信プログラムを実行し、他の生徒は受信プログラムを実行する。

### 送信プログラム

次のプログラムは、音楽をサーバーに送信するプログラムである。送信する曲は、小節ごとに6個に分けて「曲」という配列に格納している。「送信」ボタンを押すと、最初に曲の行数（6行）と送信間隔（4秒）をサーバーに送信した後、曲を一定間隔で1行ずつサーバーに送信する。実行は次の節の受信プログラムと合わせて行う。



```

サーバー！ "localhost" 接続。
曲＝配列！ 作る。
曲！ "どそそそらそー" 書く。
曲！ "ふあふあみれどー" 書く。
曲！ "そそふあふあみれー" 書く。
曲！ "そそふあふあみれー" 書く。
曲！ "どそそそらそー" 書く。
曲！ "ふあふあみれどー" 書く。
送信間隔＝4。

送信中＝ラベル！ 作る 300 45 大きさ。
送信ボタン＝ボタン！ "送信" 作る。
送信ボタン：動作＝「
    行数＝曲！ 要素数？。
    送信中！（行数）書く。
    サーバー！ "配信行数"（行数）書く。
    サーバー！ "配信間隔"（送信間隔）書く。
    タイマー！ 作る（送信間隔）間隔（行数）回数「 | 番号
    |
        メッセージ＝曲！（番号）読む。
        送信中！（メッセージ）書く。
        サーバー！ "配信"（メッセージ）書く。
    」実行。
」。
```

## 受信プログラム

次のプログラムは、音楽をサーバーから受信するプログラムである。「ダウンロード」ボタンまたは「ストリーミング」ボタンを押すと、最初に曲の行数と受信間隔を受信した後、曲を一定間隔で1行ずつ受信する。ダウンロードボタンを押したときは、曲全体を受信してから演奏する。ストリーミングボタンを押したときは、曲の一部を受信しながら随時演奏していく。

実行する際は2つのドリトルの画面で、送信プログラムと受信プログラムを実行する。そして、送信プログラムの「送信」ボタンを押した直後に受信プログラムの「ダウンロード」または「ストリーミング」のボタンを押す。

サーバー！ "localhost" 接続。

ダウンロード=ボタン！ "ダウンロード" 作る 230 45 大きさ -300 50 位置。

ストリーミング=ボタン！ "ストリーミング" 作る 230 45 大きさ -300 0 位置。

曲リスト=リスト！ 作る 200 400 大きさ 0 200 位置。

ダウンロード：動作＝「

曲=""。

曲リスト！ クリア。

行数=サーバー！ "配信行数" 読む。

受信間隔=サーバー！ "配信間隔" 読む。

ラベル！（行数）作る -50 200 位置。

時計=タイマー！ 作る（受信間隔）間隔（行数）回数「 | 番号 |

メッセージ=サーバー！ "配信" 読む。

曲リスト！（メッセージ）書く。

曲=曲！（メッセージ）連結。

」実行。

時計！ 待つ。

メロディ！ 作る（曲）追加 演奏。

」。

ストリーミング：動作＝「

曲リスト！ クリア。

行数=サーバー！ "配信行数" 読む。

受信間隔=サーバー！ "配信間隔" 読む。

ラベル！（行数）作る -50 200 位置。

時計=タイマー！ 作る（受信間隔）間隔（行数）回数「 | 番号 |

メッセージ=サーバー！ "配信" 読む。

曲リスト！（メッセージ）書く。

メロディ！ 作る（メッセージ）追加 演奏。

」実行。

」。

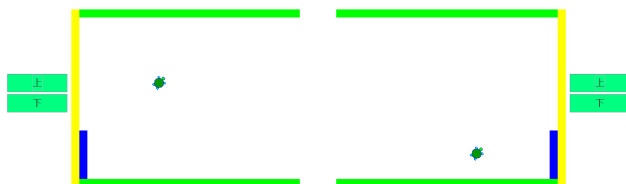
## Activity 20



# ネットワークゲームを作ろう

ネットワーク機能を使って、教室の中で友だちとゲームをするプログラムを作ってみよう。題材は、Activity 6 で作ったピンポンゲームを対戦型に拡張したものである。

作成したゲームの画面を示す。画面は左側と右側のプログラムで異なるが、内容はほとんど同じであるため、左側のプログラムを例に説明する。ピンポンゲームと変わらない部分については、Activity 6 を参照してほしい。



## 20.1 壁を作る (ステップ 1)

最初に、ゲームに登場するオブジェクトを画面に置く。上下にある長方形は、ボールを跳ね返す壁である。左の小さいほうの長方形はボールを打ち返すパドルである。大きいほうの長方形は、パドルでボールを打ち返せなかったことを知るための壁である。ボールがこの壁にぶつくと、ゲームオーバーになる。

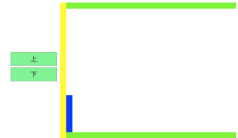
次のプログラムでは、4 個の長方形を作成している。プログラムを簡単にするために、**線の太さ**で太さ 20 の線を描くことで長方形とした。続いて、上壁を作ってから複製して下壁を作り、続いて左壁とパドルを作っている。



```
// 壁を作る (ステップ1)
カメ太=タートル! 作る。
カメ太! (緑) 線の色 20 線の太さ。
壁=カメ太! 550 歩く 図形を作る -200 200 位置。
壁! 作る -200 -220 位置。
カメ太! 90 左回り。
左壁=カメ太! (黄) 線の色 440 歩く 図形を作る -210 -230 位置。
パドル=カメ太! (青) 線の色 120 歩く 図形を作る -190 -210 位置。
```

## 20.2 パドルを動かす (ステップ 2)

画面にボタンを表示して、パドルを上下に動かせるようにする。次のプログラムでは、画面の左側に「上ボタン」と「下ボタン」という名前の2つのボタンを表示し、ボタンが押されるか上下の矢印キーが押されたときにパドルを上下に 50 ずつ移動させている。

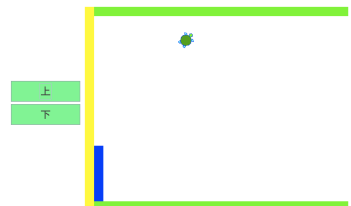


```
// パドルを動かす (ステップ2)
上ボタン=ボタン! "上" "UP" 作る -380 50 位置。
下ボタン=ボタン! "下" "DOWN" 作る -380 0 位置。
上ボタン:動作=「パドル! 0 50 移動する」。
下ボタン:動作=「パドル! 0 -50 移動する」。
```

## 20.3 ボールの動きを定義する (ステップ 3)

「カメ太」はボールの役割をする。毎回少しずつ違った位置からゲームが始まるように、横の位置は左右の中央で、縦の位置は乱数を使い、-149~150 の範囲でランダムになるようにした。壁にぶつかると、自然な角度で跳ね返る。

```
// ボールの動きを定義する (ステップ3)
カメ太! ペンなし 45 向き。
カメ太! 0 (乱数 (300) -150) 位置。
カメ太:衝突=タートル:跳ね返る。
```



## 20.4 通信を準備する (ステップ 4)

ゲームをする 2 つの画面の間では、ボールがどのような状態にあるかという情報をお互いに交換して合わせておく必要がある。たとえば、ひとつの画面でパドルと衝突して跳ね返ったら、もうひとつの画面でも同様に向きを変える必要がある。

そこで、自分のボールの状態をサーバーに書き込む「書き込み」というメソッドと、相手のボールの状態をサーバーから読み出す「読み出し」というメソッドを定義することにした。

次のプログラムで、「書き込み」の中では、現在のボール（「カメ太」）から位置と向きの情報を取り出し、それぞれ「x」、「y」、「t」という名前でサーバーに保存している。「読み出し」の中では、サーバーから位置と向きの情報「x」、「y」、「t」を取り出し、ボール（「カメ太」）にセットしている。

ステップ 4 からステップ 6 までを実行するには、サーバーと相手の画面が必要になる。詳しくはステップ 7 で説明する。

```
// 通信を準備する (ステップ4)
カメ太：書き込み＝「
    サーバー！ "x" （カメ太！ 横の位置？） 書く。
    サーバー！ "y" （カメ太！ 縦の位置？） 書く。
    サーバー！ "t" （カメ太！ 向き？） 書く
」。
```

```
カメ太：読み出し＝「
    x＝サーバー！ "x" 読む。
    y＝サーバー！ "y" 読む。
    t＝サーバー！ "t" 読む。
    カメ太！（x）（y）位置 （t）向き
」。
```

## 20.5 通信しながらボールを動かす (ステップ 5)

ステップ 5 は、このプログラムの中心的な部分である。行っている処理は画面上のボールを動かすことだが、ボールは両方の画面で同じように表示される必要があるため、サーバーを介した通

信を行うことで、ボールの状態をお互いに合わせている。

次のプログラムでは、最初に、サーバーに接続する。続いて、ステップ4で定義した「書き込み」を実行して、ボールの位置をサーバーに書き込む。そしてタイマーを作成する。ここまでが、プログラムを開始するときの動作である。

続いて、ゲームの処理を行う。本質的な動作は次の1行で記述できる。つまり、タイマーでボール役の「カメ太」を前進させる動作である。

時計！「カメ太！ 10 歩く」実行。

ここでは、他の画面とボールの状態を合わせるために、最初に「読み出し」でサーバーから相手のボールの状態を取得して、ボールを正しい位置に置く。続いてボールを移動し、移動後の新しい状態を「書き込み」でサーバーに書き込んでいる。

```
// 通信しながらボールを動かす（ステップ5）
サーバー！ "localhost" 接続。
カメ太！ 書き込み。
時計＝タイマー！ 作る 60秒 時間 0.2秒 間隔。
時計！「
    カメ太！ 読み出し。
    カメ太！ 10 歩く。
    カメ太！ 書き込み。
」実行。
```

## 20.6 ゲームの勝敗を判定する (ステップ 6)

最後に、タイマーが終了するまでゲームを続けられた場合には「ゲームクリア」というメッセージを表示する。この部分はピンポンゲームと同じであるため、詳しくは Activity 6 を参照されたい。

```
// ゲームの勝敗を判定する (ステップ6)
ゲームクリア=はい。
左壁：衝突=「：ゲームクリア=いいえ。時計！ 中断」。
時計！ 待つ。
「ゲームクリア==はい」！ なら「
    ラベル！ "ゲームクリア！ "作る (青) 文字色。
」そうでなければ「
    ラベル！ "ゲームオーバー！ "作る (赤) 文字色。
」実行。
```

ステップ 1 からステップ 6 までの全体のプログラムを掲載しておく。これは左側の画面で実行するプログラムである。

```
// 壁を作る (ステップ1)
カメ太=タートル！ 作る。
カメ太！（緑）線の色 20 線の太さ。
壁=カメ太！ 550 歩く 図形を作る -200 200 位置。
壁！ 作る -200 -220 位置。
カメ太！ 90 左回り。
左壁=カメ太！（黄）線の色 440 歩く 図形を作る -210 -230 位置。
パドル=カメ太！（青）線の色 120 歩く 図形を作る -190 -210 位置。

// パドルを動かす (ステップ2)
上ボタン=ボタン！ "上" "UP" 作る -380 50 位置。
下ボタン=ボタン！ "下" "DOWN" 作る -380 0 位置。
上ボタン：動作=「パドル！ 0 50 移動する」。
下ボタン：動作=「パドル！ 0 -50 移動する」。
```

```
// (続き)
// ボールの動きを定義する (ステップ3)
カメ太！ ペンなし 45 向き。
カメ太！ 0 (乱数 (300) -150) 位置。
カメ太：衝突=タートル：跳ね返る。

// 通信を準備する (ステップ4)
カメ太：書き込み=「
    サーバー！ "x" (カメ太！ 横の位置?) 書く。
    サーバー！ "y" (カメ太！ 縦の位置?) 書く。
    サーバー！ "t" (カメ太！ 向き?) 書く
」。
カメ太：読み出し=「
    x=サーバー！ "x" 読む。
    y=サーバー！ "y" 読む。
    t=サーバー！ "t" 読む。
    カメ太！ (x) (y) 位置 (t) 向き
」。

// 通信しながらボールを動かす (ステップ5)
サーバー！ "localhost" 接続。
カメ太！ 書き込み。
時計=タイマー！ 作る 60秒 時間 0.2秒 間隔。
時計！「
    カメ太！ 読み出し。
    カメ太！ 10 歩く。
    カメ太！ 書き込み。
」実行。

// ゲームの勝敗を判定する (ステップ6)
ゲームクリア=はい。
左壁：衝突=「：ゲームクリア=いいえ。時計！ 中断」。
時計！ 待つ。
「ゲームクリア==はい」！ なら「
    ラベル！ "ゲームクリア！ "作る (青) 文字色。
」そうでなければ「
    ラベル！ "ゲームオーバー！ "作る (赤) 文字色。
」実行。
```



## 20.7 プログラムを実行する（ステップ 7）

この節で作成しているプログラムは、サーバーや相手の画面と協調して動作するため、単体では動かない。実行するための手順を説明する。

まず、サーバーを起動する。サーバーはどのコンピュータで起動してもよいが、ここでは左側の役割をする画面を実行するコンピュータで動かすことにする。サーバーを起動するには、ドリトルの編集画面（プログラムを入力する画面）を開き、右側にある **server** をマウスでチェックする。すると、サーバーのウィンドウが画面に表示される。サーバーの画面自体は使わないので、開いたまま放しておく。

次に、右側の画面を実行するために、もうひとつドリトルの画面を起動する。右側の画面で実行するプログラムを示す。左側の画面との違いは、画面上のパドルなどが左右逆に配置されていることと、ボールの初期位置をサーバーに設定しないこと、そしてボールが両方の画面で自然な位置に表示されるように、ボールの位置を 100 だけ左にずらしてサーバーに書き込んでいることである。

このプログラムを左側の画面と同じコンピュータで実行する場合は変更の必要はないが、別のコンピュータで実行する場合には、左側の画面のコンピュータを実行しているコンピュータのホスト名または IP アドレスを調べて、プログラム中の「localhost」の部分を書き換える必要がある。

```
// 壁を作る（ステップ1）
カメ太＝タートル！ 作る。
カメ太！（緑）線の色 20 線の太さ 180 右回り。
壁＝カメ太！ 550 歩く 図形を作る 200 200 位置。
壁！ 作る 200 -220 位置。
カメ太！ 90 右回り。
左壁＝カメ太！（黄）線の色 440 歩く 図形を作る 210 -230 位置。
パドル＝カメ太！（青）線の色 120 歩く 図形を作る 190 -210 位置。
```

```
// (続き)
// パドルを動かす (ステップ2)
上ボタン=ボタン! "上" "UP" 作る 230 50 位置。
下ボタン=ボタン! "下" "DOWN" 作る 230 0 位置。
上ボタン:動作=「パドル! 0 50 移動する」。
下ボタン:動作=「パドル! 0 -50 移動する」。

// ボールの動きを定義する (ステップ3)
カメ太! ペンなし。
カメ太! 0 (乱数 (300) -150) 位置。
カメ太! 45 向き。
カメ太:衝突=タートル:跳ね返る。

// 通信を準備する (ステップ4)
カメ太:書き込み=「
  サーバー! "x" (100+ (カメ太! 横の位置?)) 書く。
  サーバー! "y" (カメ太! 縦の位置?) 書く。
  サーバー! "t" (カメ太! 向き?) 書く
」。
```

カメ太:読み出し=「  
 x=サーバー! "x" 読む。  
 y=サーバー! "y" 読む。  
 t=サーバー! "t" 読む。  
 カメ太! (x-100) (y) 位置 (t) 向き  
 」。

```
// 通信しながらボールを動かす (ステップ5)
サーバー! "localhost" 接続。
//カメ太! 書き込み。
時計=タイマー! 作る 60秒 時間 0.2秒 間隔。
時計! 「
  カメ太! 読み出し。
  カメ太! 10 歩く。
  カメ太! 書き込み。
」 実行。
```

```
// ゲームの勝敗を判定する (ステップ6)
ゲームクリア=はい。
左壁:衝突=「:ゲームクリア=いいえ。時計! 中断」。
時計! 待つ。
「ゲームクリア=はい」! なら「
  ラベル! "ゲームクリア! "作る (青) 文字色。
」 そうでなければ「
  ラベル! "ゲームオーバー! "作る (赤) 文字色。
」 実行。
```

Part X

# 付録



## 付録 A



# 授業での利用

ドリトルは小学校から大学までの幅広い範囲で利用可能である。企業の新人研修としても使われた例がある。ここでは、授業で利用するための参考として、いくつかの授業例を紹介する。

以下の授業例で、授業回数や難易度は、中学校技術・家庭や高校普通教科「情報」、大学の共通科目程度を想定している。小学校では、高学年以上であれば、多少時間を増やす程度で実施可能である。1 回あたりの授業時間は 45 分から 50 分程度を想定している。

## A.1 プログラミング

### プログラミングを体験しよう

**時間:** 2 回（高校以上では 1 回でも可能）

**スキル:** キー入力ができること

**授業の目的:** コンピュータの動く仕組みを体験する

**授業の目標:** ゲームなどのソフトウェアは人間が書いたプログラムで動いていることを理解する

**ドリトル:** オンライン版、ローカル版

**学習環境:** コンピュータ室。教員 1 名

**授業内容:** タートルの移動とアニメーションまで

- 1 タートルグラフィックス（Activity 1）
- 2 タイマー（Activity 4）

### ソフトウェアの仕組みを学ぼう

**時間:** 3 回～4 回

**スキル:** キー入力ができること

**授業の目的:** コンピュータの動く仕組みを体験する

**授業の目標:** ゲームやメールなどのソフトウェアは人間が書いたプログラムで動いていることを理解する

**ドリトル:** オンライン版（チャットの実習は不可）、ローカル版

**学習環境:** コンピュータ室。教員 1 名

**授業内容:** 対話的な操作とネットワークまで

- 1 タートルグラフィックス (Activity 1, Activity 2)
- 2 タイマーによるアニメーション (Activity 4)
- 3 ペイントソフト (Activity 3)
- 4 チャットプログラム (Activity 17, Activity 18)

## ゲームを作ろう

**時間:** 9 回

**スキル:** キー入力ができる

**授業の目的:** プログラミングの体験

**授業の目標:** ゲームやアニメーションのような作品プログラムを作ることができる

**ドリトル:** オンライン版、ローカル版

**学習環境:** コンピュータ室。教員 1 名

**授業内容:** 繰り返し、ボタン、衝突など

- 1 ドリトルを使ってみよう、プログラムを書いてみよう (Activity 1)
- 2 繰り返し、メソッドの定義、図形 (Activity 1, Activity 2)
- 3 ボタン、タイマー、ラベル (Activity 3, Activity 4)
- 4 衝突 (Activity 5)
- 5, 6 分岐、ゲーム (Activity 6)
- 7, 8 作品制作
- 9 相互評価・発表

## A.2 ネットワークプログラミング

### チャットの仕組み

**時間:** 4 回

**スキル:** キーボードから入力できること。

**授業の目的:** ネットワークの存在に気づき、通信をする楽しさを味わう

**授業の目標:** ネットワークではソフトウェア同士が通信していることを体験的に理解する

**ドリトル:** ローカル版

**学習環境:** 生徒のコンピュータが LAN で接続されているコンピュータ室、教員 1 名

**授業内容:**

- 1 IP アドレス、サーバーへのオブジェクトの読み書き (Activity 17)
- 2 音楽の配信 (Activity 19)
- 3 1 行のメール (Activity 18)
- 4 複数でチャット (Activity 18)

## A.3 ロボット制御

### ロボットでダンスを踊ろう

**時間:** 4 回

**スキル:** キー入力ができること

**授業の目的:** ロボットや家電製品のような組み込み機器はコンピュータが制御していることを理解する

**授業の目標:** プログラムによってロボットの移動を制御できる

**ドリトル:** ローカル版

**学習環境:** コンピュータ室とロボットを走らせる場所 (廊下など)。教員 1 名

**授業内容:** ロボットの移動命令まで

- 1 ロボットを動かすための命令を書こう (Activity 13)
- 2~4 ロボットでダンスを踊ろう! (Activity 14)

### ロボットで迷路脱出

**時間:** 6 回

**スキル:** 「ロボットでダンスを踊ろう」の内容を理解していること

**授業の目的:** 制御では外部からの入力を利用されることを理解する

**授業の目標:** タッチスイッチを利用して壁への衝突を検知するプログラムを作成する

**ドリトル:** ローカル版

**学習環境:** コンピュータ室と迷路コース。教員 1 名

**授業内容:** タッチスイッチの入力まで

- 1~4 迷路でゴールにたどり着けるか挑戦しよう (Activity 13, Activity 14)
- 5~6 タッチスイッチを付けたロボットで挑戦しよう (Activity 15)

## ロボットで物を運ぼう

**時間:** 11 回

**スキル:** 「ロボットで迷路脱出」の内容を理解していること

**授業の目的:** 物を運ぶ作業を行えるロボットの仕組みを理解し、プログラムを作成できるようになる

**授業の目標:** 身近な機械でセンサーが使われていること。そして、構造化やメソッドでプログラムを部品化することの大切さを理解する

**ドリトル:** ローカル版

**学習環境:** コンピュータ室と競技コース。教員 1 名

**授業内容:** 3 軸ロボットの制御

**1~2** 玉入れロボットの製作。すでに製作していた 2 軸ロボットを拡張する

**3~4** 玉入れロボットの動作確認

**5~7** 繰り返しとセンサーを利用したプログラム (Activity 15)

**8~9** タッチセンサーを利用してアームを動かす。繰り返しからの脱出 (Activity 16)

**10~11** 物を運ぶロボットの完成 (Activity 16)

## 付録B



# よいプログラムを書くために

文章に分かりやすい（よい）文章と理解しづらい（悪い）文章があるように、プログラムにもよいプログラムと悪いプログラムがある。ここでは、よいプログラムを書くためのヒントを説明する。

## B.1 よいプログラムって何だろう

よいプログラムとは、簡単にいうと、「目的を達成するプログラム」のことである。ただし、目的は人によって、またはそのときどきによって違ってくる。ここでは、次の2つの場合を考えてみる。

### (a) 決められたプログラムを作る

授業で指示された課題のプログラムを作ったり、仕事で注文を受けてプログラムを作ったりする場合が相当する。「このようなプログラムを作ってください」という、はっきりした要求が与えられる。それを満たすプログラムを作れば、目的を達成したことになる。

### (b) 自由に考えてプログラムを作る

授業で自由作品の課題プログラムを作ったり、自分で目的を決めてプログラムを作ったりする場合が相当する。どんなプログラムを作るかはある程度自由に決められる。自分で作りたいと思ったプログラムを作れば、目的を達成したことになる。

## B.2 よいプログラムの作り方

プログラムは自由に作ることができるが、いくつかのコツを知っていると、目的のプログラムを確実に素早く作ることができる。ここでは、その中でももっとも基本的なコツを選んで伝授していく。



## (1) 作るプログラムをイメージする

そもそもどんなプログラムを作ればよいかが分かっていないと、どこから作り始めてよいかも分からない。「完成したら、どのような画面になって、どのような動きをするのか」ということを考えてみよう。このとき、できるだけ具体的にイメージすることが大切である。頭の中で、そのプログラムが動く様子が生き生きとイメージできたり、動いている様子をノートなどに絵で書けるように考えてみよう。

## (2) どんな機能が必要かを考える

最初は簡単に作れそうに思ったプログラムでも、書いているうちにどんどん長くなってしまふことがある。長くなってから全体を理解しようとしても、何十行もあると、自分が書いたプログラムなのに理解できない。

そこで、大まかに 3~5 個くらいで、作りたいプログラムに必要な機能を考えてみるとよい。機能というのは、そのプログラムが行える動作のこと。最初は簡単でないかもしれないが、どのような順番でプログラムを作っていくかを考えることは重要である。

どのような機能を考えればよいかは、Part III で紹介したいいくつかのゲームが参考になる。たとえば、Activity 5 の「宝物拾いゲーム」は、次の 4 個のステップで作られていた。そして、いちどに全体を作るのではなく、ステップ 1 から順に小さなプログラムのまとまりごとに作っていくことで、最終的にゲームを作り上げることができた。プログラムのまとまりの先頭には、何をする機能を **コメント**（「//」）で書いておくとよい。

### (ステップ 1) タートルを操作する

最初は、画面の上に目に見えるオブジェクトを置くプログラムから作るのがよい。このゲームでは、画面に主役のタートルとボタンを作った。プログラムの結果が画面に現れるので、自分のプログラムが正しく動いていることを確認できる。

### (ステップ 2) タートルを前進させる

次に、画面のオブジェクトに動きを付ける。ステップ 2 では、主役のタートルを前進させていた。ドライブゲームとして遊ぶことができる。

### (ステップ 3) 宝物を画面に置く

続くステップでは、脇役のオブジェクトを登場させたり、衝突したときの動作を定義したりして、プログラム全体を完成させていく。ステップ 3 では、脇役である宝物のオブジェクトを画面に置いていた。

### (ステップ 4) 宝物を拾う

最後のステップでは足りない機能を作り、プログラムを完成する。ステップ 4 では宝物とタートルが衝突したときの動作を定義した。

### (3) 1行ずつ実行しながら作っていく

プログラムがうまく動かないときに、長いプログラムの中でどこに原因があるのかを調べるのは簡単ではない。たとえば、いちどに 10 行のプログラムを入力して実行すると、何らかのエラーが表示されるか、思ったような動きをしてくれないことが多い。

しかし、「この行に問題がある」ということが分かれば、原因を調べることはそう難しくない。そこで、プログラミングに慣れるまでは、プログラムは基本的に 1 行入力するごとに実行して動作を確かめながら進めることが望ましい。

実行しながらプログラムを書いていくことは、プログラムが正しいことを確認できることに加えて、「少しずつ形になっている」ことを実感できるので、プログラムを作る作業を楽しくする効果がある。プログラミングは本来楽しい作業なので、それを早いうちに体験できると速く上達することにもつながる。

もし実行してエラーが表示された場合には、そのメッセージを手がかりに問題の箇所を探すことになる\*1。こまめに実行して確認していれば、前回実行した部分までは正しく動いていたので、問題は直前に入力した数行にあることが分かる。必要に応じて、問題のありそうな行の先頭に「//」を書いてコメントにすることで、その行にエラーがあるのかを調べることもできる。「この行までは動く」「この行を加えると動かない」という切り分けができれば、問題はずっと見つけやすくなる。

### (4) 目標を達成できたか確認しよう

エラーがなくなって動くようになったら、プログラムが完成したのかどうか、そして足りない部分があればどうすればよいかを考えよう。確認すべき点はプログラムごとに異なるが、次の点はほぼ共通に求められていると考えることができる。

**(目的を達成)** 与えられた課題を達成しているか

作りたかったものができているかを確認する。

**(正しく動く)** 実行すると正しく動くか

実行するとエラーになったり、途中で動きがおかしくなることはないか確認する。

**(読みやすい)** 他の人が読んで理解できるか

一週間後の自分が読んでも分かるように、そして先生など他の人が読んでも理解できるように、プログラムを整理しておく。機能ごとにまとまりを作ったり、機能を説明するコメントを書いておくことは重要である。

自由課題の作品や、自分で目的を決めて作るプログラムでは、次の点も確認するとよい。

---

\*1 エラーメッセージとデバッグについては Activity 1 のコラムを参照。

**(面白さがある)** 見る人に楽しんでもらえるか

他の人に見せたり使ってもらうプログラムでは重要になることがある。

**(独自の工夫点)** 自分なりの工夫点があるか

図形の形やボタンの配置など、ちょっとしたことでもいいので工夫した点があると達成感につながる。

**(画面デザイン)** 見た目も大切

他の人に見せたり使ってもらうプログラムでは、見栄えや使いやすいデザインも重要になる。せっかく作るのだから、使いやすかつこい作品に仕上げよう。

## 付録C



# 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。

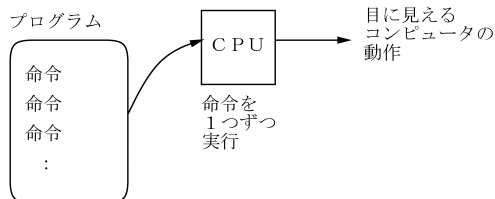
## C.1 プログラミングとは

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。

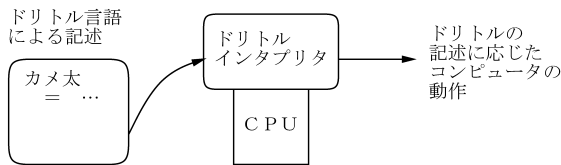


そして、プログラミングとは「プログラムを作ること」であり、上の説明の延長でいえば、コンピュータに「自分がさせたいこと」を実行させるように、命令を並べて行く作業だ、ということになる。

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」の組み合わせでプログラムを作るのは大変複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。

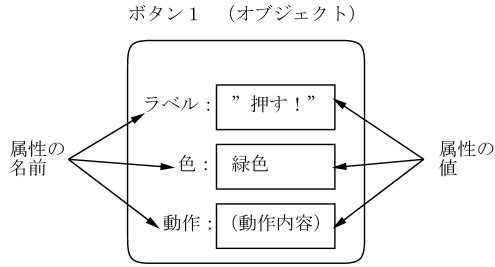
プログラミング言語で書き表したものを実際に実行するためには、また別のソフトウェア（プログラミング言語処理系）を使う。プログラミング言語処理系には、プログラミング言語で書き表したものを機械語に変換する方式（コンパイラ）と、プログラミング言語で書き表したものを読み取りながら直接その動作を実行する方式（インタプリタ）とがある。



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。

## C.2 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の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。

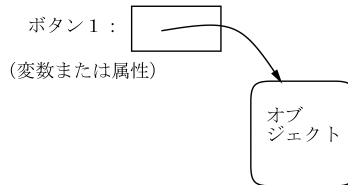


このように、すべての情報をオブジェクトとして統一的に扱い、さまざまな機能もオブジェクトに付随しているという形で扱うプログラミング言語を**オブジェクト指向言語**という。今日のソフトウェア開発ではオブジェクト指向言語が多く使われる。C++、Java などはオブジェクト指向言語の例である。ドリトルもオブジェクト指向言語である。

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な内容（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くて、ずっと1つのオブジェクトを入れておく場合には、その変数の名前を、その「オブジェクトの名前」だと思っても差し支えない。

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。

## C.3 メッセージ送信

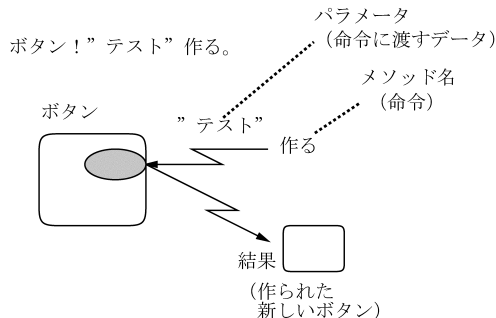
前節で、オブジェクトに付随するもののなかに「機能」があると述べた。これらの機能も、オブジェクトにプロパティとして付属しているので、そのプロパティ名を用いて「この機能」ということを指定できる。オブジェクト指向言語では通常、機能のことを**メソッド**と呼ぶので、本書でも以下こちらの呼び方を用いる。メソッドとは要するにオブジェクトが持つ（オブジェクトに付随する）機能のことだと覚えておいて頂ければよい。オブジェクトのメソッドを使うということは、通常はそのメソッドを「呼び出す（動かす、働かせる）」ことである。

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。

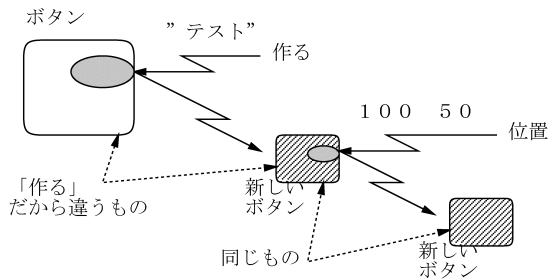
すべてのメソッドは、実行した後、何らかの値（オブジェクト）を返す。1つのメッセージ送信（メソッド呼び出し）の後に、続けてまた別のメソッド名を書くことで、前のメソッドが返したオブジェクトに対するメソッド呼び出しを行うことができる。これをメッセージの**カスケード**（直列接続）と呼ぶ。

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「(x)」などのように指定する（C.4節で詳しく説明する）。

ボタン! "テスト" 作る 100 50 位置。



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」

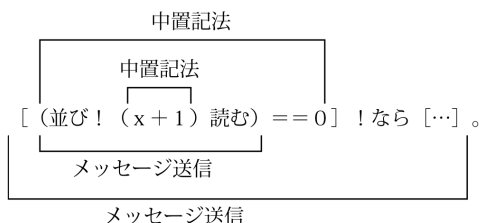


という名前のメソッドと、「？」で終る名前を持つオブジェクトのさまざまな性質を調べることが目的のメソッドは、この原則の例外になっている。

## C.4 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「 $x + 1$ 」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「 $x$  ! 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、「 $(\dots)$ 」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「 $(\dots)$ 」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「 $x + y$ 」をメッセージ送信記法で書くと「 $x$  !  $(y)$  足す」になるが、この「 $(y)$ 」は丸かつこの中に「!」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「 $(\dots)$ 」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。

## C.5 ブロックとメソッド

角っこ（「…」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。

```
x = 10。
動作1 = 「x = x + 10」。
動作1！ 実行。
動作1！ 実行。
ラベル！（x）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：x = 10。
ボタン1：動作 = 「x = x + 10。ボタン1！（x）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  $x$  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。

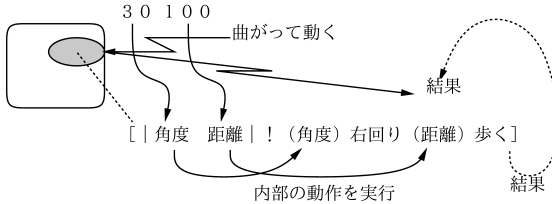
ブロックがメソッドになっているときは、その中では、 $x$  等はそのメソッドを持っているオブジェクトのプロパティ  $x$  に対応する（C.10 節で詳しく説明する）。ここでは、まず  $x$  に格納されている値に 10 を足した値を計算し、その結果を再び  $x$  に格納している。次に、ボタンの位置を  $X$  座標が  $x$  の値、 $Y$  座標が 50 になるように変更している。これにより、ボタンは押されるごとに位置が右に 10 ずつ移動することになる。

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 C.10 節で詳しく説明する）

```
カメ太 = タートル！ 作る。
カメ太：曲がって動く = 「| 角度 距離 |！（角度）右回り（距離）歩く」。
```

x = カメ太! 30 100 曲がって動く 横の位置?。  
ラベル! (x) 作る。

「!」の前に何も無いときは、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、つまりカメ太自身が「曲がって動く」の返す値となる。そこで引き続きカメ太に対するメソッド「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。

## C.6 ブロックと制御構造

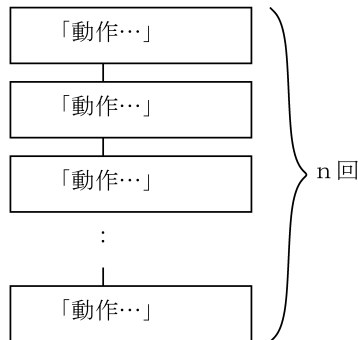
ドリトルのブロックは、メソッドを作ることのほかに、さまざまな制御構造を実現するのにも使われる。それには、ブロックオブジェクト自身が持つメソッドを使うのが基本である。以下で説明する。

### 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。

「動作…」！ 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。

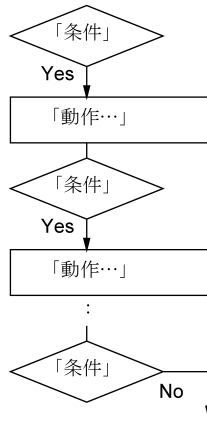
ラベル！ (a) 作る。

## 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。

「条件」！の間「動作…」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、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

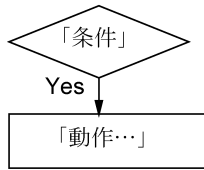
条件分岐（枝分かれ）は、ある条件の成否に応じてプログラムの一部を実行する。条件分岐もブロックのメソッドならとそうでなければと実行を組み合わせで記述する。

「…」！ なら 「…」 実行。

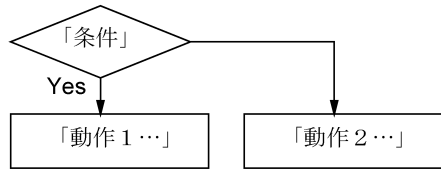
「…」！ なら 「…」 そうでなければ 「…」 実行。

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。

「条件」！なら「動作…」実行



「条件」！なら「動作1…」  
そうでなければ「動作2…」実行



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。

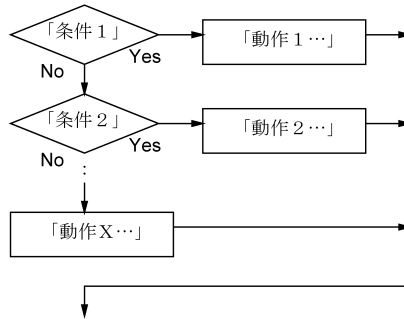
ラベル！（結果）作る。

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。

「条件 1」！なら「動作 1…」  
 そうでなければ「条件 2」なら「動作 2…」  
 そうでなければ「条件 3」なら「動作 3…」  
 そうでなければ「動作 X」実行



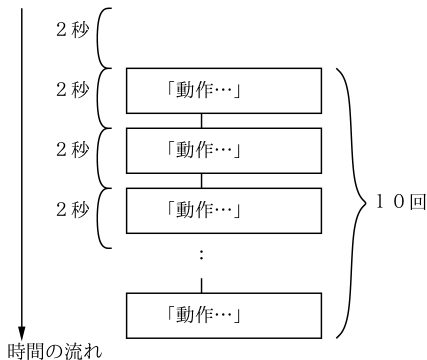
## C.7 タイマーとスレッド

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。

時計＝タイマー！作る 2 間隔 10 回数。  
 時計！「動作…」実行。



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。

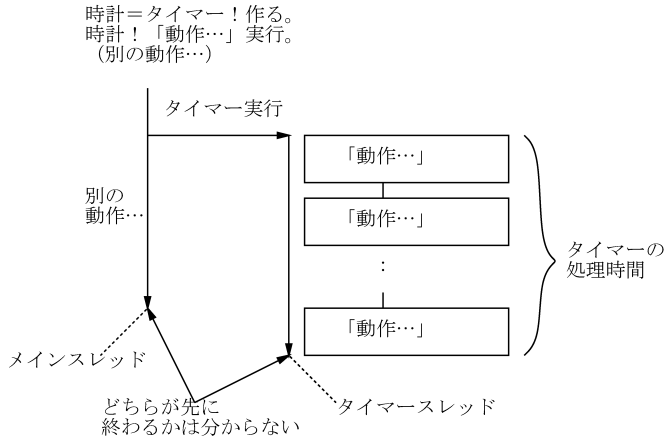
```
表示＝ラベル！ "はるー" 作る。  
時計＝タイマー！ 作る。  
時計！ 「|i| 表示！ (i) 0 移動する」実行。
```

## スレッドと待ち合わせ

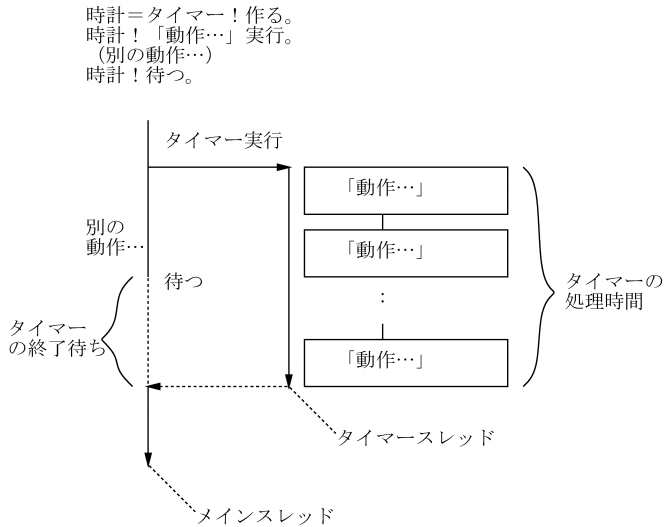
ドリトルのプログラムは通常、上から順に実行され、ひとつの文の実行が終るのを待って次の文の実行が行われる。

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く。このように、ひとつのプログラムの中で同時に複数の処理を並行して実行する仕組みを**スレッド**と呼ぶ。通常のプログラムも、1 つのスレッド（**メインスレッド**）を使って実行されている。



タイマースレッドとメインスレッドは並行に動作し、どちらが先に終わるかはプログラムの内容によってさまざまだが、ときにはタイマーの実行が終了するのを待ってから次に進みたいことがある。このようなときには、**待つ**でタイマーの終了を待つことができる。



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！「|i| 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、最後の行のプログラムが実行され、しばらく時間が経ってから画面に「こんにちは」というメッセージが表示される。

```
表示=ラベル！ "はるー" 作る。
時計=タイマー！ 作る。
時計！ 「|i| 表示！ 3 -2 移動する」実行。
時計！ 待つ。
ラベル！ "こんにちは"作る。
```

## C.8 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。
```

1	2	3
123	456	789

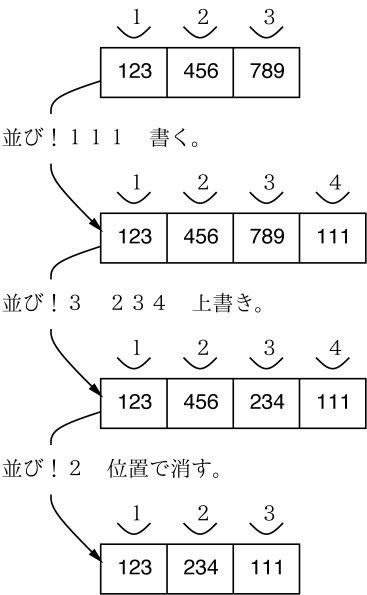
配列に入っている要素の数は、**要素数？** で調べることができる。次のプログラムでは、配列に 3 個の数値を入れた後で、要素数を画面に表示している。画面には 3 が表示される。

```
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（要素数は変わらず）。

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。

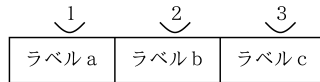
並び！（ラベル！ "c" 作る 0 0 位置）書く。

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。

」。



並び！「| 並び要素 | 並び要素！ 10 0 移動する」それぞれ実行。



ラベル a！ 10 0 移動する。

ラベル b！ 10 0 移動する。

ラベル c！ 10 0 移動する。

## C.9 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は不可欠である。

### 「作る」と親子関係

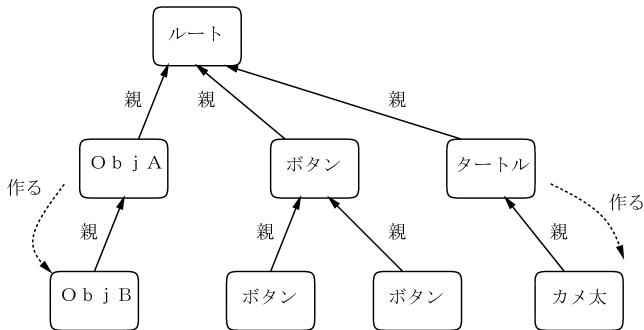
ドリトルのすべてのオブジェクトは親子関係を持っている。あるオブジェクトに対して、親のオブジェクトのことを**プロトタイプオブジェクト**と呼ぶこともある。

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。

ObjB=ObjA！作る。



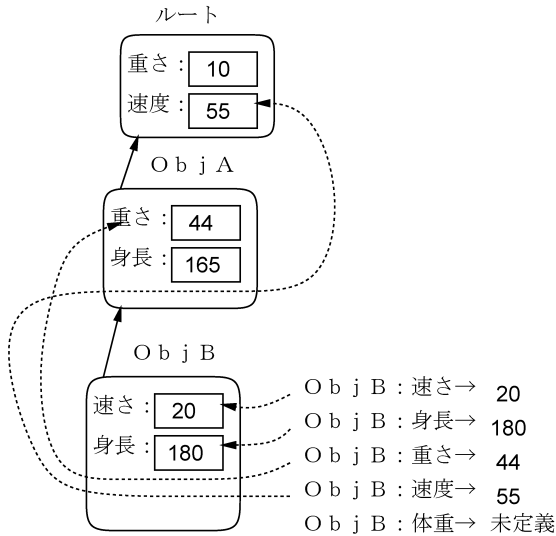
子のオブジェクトは親のオブジェクトが持っていたプロパティ（値やメソッド定義）をすべてそのまま引き継ぎ、親と同じように扱うことができる。この仕組みを利用して、タートル、ボタン、配列など、さまざまな標準オブジェクトをプロトタイプとして持つオブジェクトを必要なだけ作って利用できるわけである。

## 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。

Obj A=ボタン！ 作る。  
 Obj B=Obj A！ 作る。  
 Obj B: 身長=180。  
 Obj A: 身長=165。  
 ラベル！ (Obj B: 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。

## C.10 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名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。

```
x = 100。
ルート：x = 100。
：x = 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。

```
ObjB = ボタン！ 作る。
ObjB : x = 200。
ObjB : 増加 = 「|d| x = x + d」。
ObjB ! 21 増加。
ラベル！ (ObjB : x) 作る。
```

ここで注意すべきこととして、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。

```
ObjB = ボタン！ 作る。
x = 100。
ObjB : 増加 = 「|d| : x = : x + d」。
ObjB ! 21 増加。
ラベル！ (ObjB : x) 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。

```
Obj B=ボタン！ 作る。
Obj B：距離=「|x1 y1 x2 y2 ; dx dy|
    dx = x1 - x2. dy = y1 - y2。
    sqrt (dx*dx+dy*dy)」。
d=Obj B！ 100 100 200 200 距離。
ラベル！（d）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<sup>\*1</sup>。

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。

## C.11 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

<sup>\*1</sup> 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。

る。その記法と意味は次の通り:

$X ::= \alpha$	書き方 $\alpha$ を $X$ と呼ぶ、ないし $X$ の定義。
$\alpha \mid \beta$	$\alpha$ または $\beta$ 。
$[\alpha]$	$\alpha$ または空。
$\alpha \cdots$	$\alpha$ が 1 個以上並んだもの。
$(\alpha)$	$\alpha$ 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 | ひらがな | 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、ひらがな、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、ひらがな、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（文字の並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   A B 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 = [x = x + 10. ボタン1、(x) 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
カメ太=タートル! 作る。 // カメ太を作る
// カメ吉=タートル! 作る。
```

## 付録D



# 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、配列については付録C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、本文の各章と付録Eを参照されたい。

## D.1 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。

## D.2 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で計算しているため、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「**×**」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。

## D.3 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 "abc xyz" 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るし、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「こんにちはカメ太さん」という文字列が表示される。

```
x="こんにちは"。
y="カメ太さん"。
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、文字列の一部分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。
```

## D.4 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こと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。

## D.5 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。

## D.6 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）のには「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、前進していつて何かにぶ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」。  
カメ太＝タートル！ 作る。  
カメ吉＝タートル！ 作る 100 0 位置。  
タイマー！ 作る「カメ太！ 10 歩く」実行。
```

## D.7 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！ 2 1 拡大する」。  
カメ太＝タートル！ 作る。
```

さんかく = 「カメ太！ 100 歩く 120 左回り」！ 3 繰り返す（赤）図形を作る。  
さんかく！ 変形。

## D.8 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、カメ太は指定された歩数だけ前進する。

```
カメ太=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=「カメ太！（窓！ 読む） 歩く」。
```

フィールドでは、**リターンキー**（Enter キー、)を押すことで、ボタンを押したときのように入力の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、カメ太は指定された歩数だけ前進する。

```
カメ太=タートル！ 作る。
窓=フィールド！ 作る。
窓：動作=「カメ太！（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。
```

## D.9 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="カメ太"。
友人=配列! "カメ吉" "カメ子" 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。

## D.10 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="カメ太です"。
```

```
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" "カメ吉です" 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。
```



# ドリトルの命令一覧

## E.1 基本オブジェクト

### 数値

- 数値を表すオブジェクトです。
- 代入式の右辺や、括弧で囲まれた部分に数式を**中置記法**で記述できます。
- 数式で扱えるデータは、数値と、数値に変換できる値を持つ文字列です。
- 2進(0b)と16進(0x)で0b1100、0xFFのように定数を記述できます。
- 角度は1周を360度とする角度で表します。
- 計算はJavaの倍精度実数(double)で行われます。ただし、一部の関数演算は単精度実数(float)で行われることがあります。
- 定数として、**円周率**を表す $\pi$ 、**PI**が用意されています。
- 数値演算子「+, -, \*, /」は、内部的にそれぞれ「add, sub, mul, div, mod」という命令に変換されて扱われます。論理演算子「==, !=, >, >=, <, <=」も、内部的にそれぞれ「eq, ne, gt, ge, lt, le」という命令に変換されて扱われます。

**+, -, \*, ×, /, ÷** : 四則演算。

(例) 「3 \* 40」を計算し「120」を表示します。

**ラベル！ (3 \* 40) 作る。**

**足す (add)**, **引く (sub)**, **掛ける (mul)**, **割る (div)** : 四則演算。命令として使います。

(例) 「3 \* 40」を計算し「120」を表示します。

**ラベル！ (3! 40 掛ける) 作る。**

**%**: 余り

(例) 8を3で割った余りを計算し「2」を表示します。

**ラベル！ (8 % 3) 作る。**

**余り (mod)** : 余り。命令として使います。



(例) 8 を 3 で割った余りを計算し「2」を表示します。

**ラベル！ (8！ 3 余り) 作る。**

**==, !=, #, >, >=, ≥, <, <=, ≤** : 比較演算。両辺が数値または数値に変換できる文字列の場合は、数値として比較されます。

(例) 「4 > 3」を計算し「[true]」を表示します。

**ラベル！ (4 > 3) 作る。**

**sqrt**: ルート ( $\sqrt{\quad}$ )

(例) 「 $1 + \sqrt{4}$ 」を計算し「3」を表示します。

**ラベル！ (1 + sqrt (4)) 作る。**

**sin, cos, tan**: 三角関数。

(例) 「sin(30)」を計算し「0.5」を表示します。

**ラベル！ (sin (30)) 作る。**

**asin, acos, atan, atan2**: 三角関数の逆関数。

(例) 「arcsin(0.5)」を計算し「30」を表示します。

**ラベル！ (asin (0.5)) 作る。**

atan2 は、X 座標の値に Y 座標の値をパラメータとして実行します。atan は  $-90 \sim 90$  の値を返しますが、atan2 は  $-180 \sim 180$  の値を返します。

(例) 「(-10, 10)」の座標から正接の逆関数を計算し「135」を表示します。

**ラベル！ (-10！ 10 atan2) 作る。**

**round, ceil, floor**: 丸め、切り上げ、切り捨て。

(例) 「0.7」を四捨五入し「1」を表示します。

**ラベル！ (round (0.7)) 作る。**

**exp**: 指数関数。

(例) 「 $e^{0.5}$ 」を計算し「1.6487212」を表示します。

**ラベル！ (exp (0.5)) 作る。**

**log**: 底が 10 の対数。

(例) 「log 100」を計算し「2」を表示します。

**ラベル！ (log (100)) 作る。**

**ln**: 底が e の対数。

(例) 「ln 100」を計算し「4.6051702」を表示します。

**ラベル！ (ln (100)) 作る。**

**pow**: べき乗。「2 の 3 乗」は「pow (2,3)」ではなく、「2！ 3 pow」と書くことに注意してください。

(例) 「 $2^3$ 」を計算し「8」を表示します。

**ラベル！ (2！ 3 pow) 作る。**

**abs**: 絶対値。

(例) 「`| - 3 |`」を計算し「3」を表示します。

**ラベル！ (abs (-3)) 作る。**

**乱数, random:** 正の整数を与えると、実行するたびに値が異なる 1～n の整数を返します。

(例) 1 から 10 までの整数をランダムに表示します。

**ラベル！ (random (10)) 作る。**

0 か負の数を与えた場合には、実行するたびに値が異なる 0～1 の実数を返します。

(例) 0 から 1 までの実数をランダムに表示します。

**ラベル！ (random (0)) 作る。**

**乱数初期化:** 0 以外の整数を与えると、それ以降に生成される乱数が毎回同じ順序で生成されるようになります。0 を与えると、そのときどきのランダムな順序に戻ります。

(例) 整数「5」に対応した乱数系列を表示します。

**乱数初期化 (5)。**

**ラベル！ (random (10)) 作る。**

**進数:** n 進数に変換します。n は 2～16 の整数です。

(例) 「10」の 2 進表現である「1010」を表示します。

**ラベル！ (10 ! 2 進数) 作る。**

**大きい整数にする:** 多倍長整数オブジェクトを作ります。10 桁以上の値を生成する場合は、数値でなく文字列から生成してください。

(例) 値が 2 の多倍長整数オブジェクトを作り、「 $2^{70}$ 」を計算します。

**x = 2 ! 大きい整数にする。**

**ラベル！ (x ! 70 pow) 作る。**

**コード文字:** 指定された文字コード (UTF-16) の文字を返します。

(例) 文字列「"A"」を表示します。

**ラベル！ (0x41 ! コード文字) 作る。**

(例) 文字列「"あ"」を表示します。

**ラベル！ (0x3042 ! コード文字) 作る。**

## 多倍長整数

- 長い桁数の整数を扱うオブジェクトです。
- 文字列 (または数値) に「大きい整数にする」を送り生成します。
- 数値 (内部は倍精度実数) から生成する場合は、初期値が数値の有効数字に制限されます。そのため、10 桁以上の場合は数値でなく文字列から生成してください。
- (例) 文字列から多倍長整数を生成します。

`x = "1000000" !` 大きい整数にする。

- 数値演算子「`+`」, 「`-`」, 「`*`」, 「`/`」, 「`%`」は、内部的にそれぞれ「`add`」, 「`sub`」, 「`mul`」, 「`div`」, 「`mod`」という命令に変換されて扱われます。論理演算子「`==`」, 「`!=`」, 「`>`」, 「`>=`」, 「`<`」, 「`<=`」も、内部的にそれぞれ「`eq`」, 「`ne`」, 「`gt`」, 「`ge`」, 「`lt`」, 「`le`」という命令に変換されて扱われます。

**`+`**, **`-`**, **`*`**, **`x`**, **`/`**, **`÷`** : 四則演算

(例) 「`3 * 40`」を計算し「120」を表示します。

**`x = 3 !` 大きい整数にする。**

**ラベル ! (`x * 40`) 作る。**

**足す (`add`)**, **引く (`sub`)**, **掛ける (`mul`)**, **割る (`div`)** : 四則演算。命令として使います。

(例) 「`3 * 40`」を計算し「120」を表示します。

**`x = 3 !` 大きい整数にする。**

**ラベル ! (`x ! 40 掛ける`) 作る。**

**`%`**: 余り

(例) 8 を 3 で割った余りを計算し「2」を表示します。

**`x = 8 !` 大きい整数にする。**

**ラベル ! (`x % 3`) 作る。**

**余り (`mod`)** : 余り。命令として使います。

(例) 8 を 3 で割った余りを計算し「2」を表示します。

**`x = 8 !` 大きい整数にする。**

**ラベル ! (`x ! 3 余り`) 作る。**

**`==`**, **`!=`**, **`≠`**, **`>`**, **`>=`**, **`≥`**, **`<`**, **`<=`**, **`≤`** : 比較演算。両辺が多倍長整数または多倍長整数に変換できる文字列の場合は、多倍長整数として比較されます。

(例) 「`4 > 3`」を計算し「`[true]`」を表示します。

**`x = 4 !` 大きい整数にする。**

**`y = 3 !` 大きい整数にする。**

**ラベル ! (`x > y`) 作る。**

**`pow`**: べき乗。「2 の 3 乗」は「`pow (2,3)`」ではなく、「`2 ! 3 pow`」と書くことに注意してください。

(例) 「`270`」を計算します。

**`x = 2 !` 大きい整数にする。**

**ラベル ! (`x ! 70 pow`) 作る。**

**`abs`**: 絶対値

(例) 「`| -3 |`」を計算し「3」を表示します。

**`x = 3 !` 大きい整数にする。**

**ラベル！ (abs (x)) 作る。**

**コード文字:** 指定された文字コード (UTF-16) の文字を返します。

(例) 文字列「"A"」を表示します。

**x = 0x41 ! 大きい整数にする。**

**ラベル！ (x) 作る。**

(例) 文字列「"あ"」を表示します。

**x = 0x3042 ! 大きい整数にする。**

**ラベル！ (x) 作る。**

## 文字列

- 文字の並びを扱うオブジェクトです。
- 数値を表す文字列 (例: "123.45") は数値として扱うことができます。
- 任意の文字は数値の「コード文字」命令を使って文字コードから文字を生成できるほか、文字列を囲む「"」などの引用符を表す文字の定数が用意されています。

**" : dq, ダブルクオート, ダブルクォーテーション**

**“ : ldq, 左ダブルクオート, 左ダブルクォーテーション**

**” : rdq, 右ダブルクオート, 右ダブルクォーテーション**

**『 : ldb, 左二重かぎ括弧**

**』 : rdb, 右二重かぎ括弧**

- 「含む?」「分割」「置き換える」「全部置き換える」では、正規表現で文字列のパターンを指定できます。使用できる主な表現を示します。<sup>\*1</sup>

**.** : 任意の 1 文字。

**?** : 直前の文字の 0 回か 1 回の繰り返し。

**\*** : 直前の文字の 0 回以上の繰り返し。

**+** : 直前の文字の 1 回以上の繰り返し。

**^** : 先頭。

**\$** : 末尾。

**|** : 左右のパタンのいずれか (OR)。

**( )** : 範囲指定。

**[ ]** : 列挙された文字のいずれか。「-」は文字範囲。先頭の「^」は続く文字を含まないという指定。

**==, !=, #, >, >=, ≥, <, <=, ≤** : 比較演算。両辺が数値または数値に変換できる文字列の

<sup>\*1</sup> 正規表現の詳細は、市販の書籍や Web サイトの解説などを参照してください。マッチした文字列の参照はサポートしていません。

場合は、数値として比較されます。それ以外は文字列として比較されます。

(例) 「"b" > "a"」を計算し「[true]」を表示します。

**ラベル！ ("b" > "a") 作る。**

**実行:** 文字列をドリトルのプログラムとみなして実行します。

(例) 文字列「"カメ太=タートル！ 作る 100歩 歩く。"」をプログラムとして実行します。

**"カメ太=タートル！ 作る 100歩 歩く。"！ 実行。**

**+**：文字列を連結します。

(例) 2つの文字列「"私は"」と「"カメ太です"」を連結し「"私はカメ太です"」を表示します。

**ラベル！ ("私は" + "カメ太です") 作る。**

**連結:** 文字列を連結します。複数の文字列を連結できます。

(例) 3つの文字列「"私は"」と「"カメ太"」と「"です"」を連結し「"私はカメ太です"」を表示します。

**ラベル！ ("私は"！ "カメ太" "です" 連結) 作る。**

**部分:** 文字列を切り出します。「m n 部分」で m 文字目から n 文字を取り出します。

(例) 文字列「"私はカメ太です"」の 3 文字目から 5 文字を切り出し「"カメ太です"」を表示します。

**ラベル！ ("私はカメ太です"！ 3 5 部分) 作る。**

**長さ？**：文字数を返します。

(例) 文字列「"はろー"」の長さを計算し「3」を表示します。


**ラベル！ ("はろー"！ 長さ?) 作る。**

**何文字目？**：文字列が何文字目に含まれるかを調べます。含まれない場合は 0 を返します。

(例) 文字列「"カメ太"」が「"私はカメ太です"」の何文字目に含まれているかを調べ「3」を表示します。

**ラベル！ ("私はカメ太です"！ "カメ太" 何文字目?) 作る。**

**含む？**：文字列を含むかを判定します。真偽値が返ります。文字列には正規表現を使えます。

(例) フィールドに文字列を入力して (Enter キー, ) を押すと、文字列に「山」という文字が含まれる場合はフィールドに「はい」が表示されます。

**f=フィールド！ 作る。**

**f: 動作= 「[s] 「s! "山" 含む?」! なら 「f! "はい" 書く」 実行」。**

**分割:** 区切り文字列を指定すると、分割した文字列が入った配列を返します。区切り文字列には正規表現を使えます。

(例) 文字列「"I/am/kameta"」を区切り文字「"/"」で分割し、「"I"」、「"am"」、「"kameta"」を要素とする配列「結果」を作り表示します。

**結果** = "I/am/kameta" ! "/" 分割。

**ラベル** ! (結果) 作る。

**置き換える** : 文字列の一部を置き換えた文字列を返します。元の文字列は変更されません。

置き換えは 1 回だけ行われます。置き換える文字列の指定には正規表現を使えます。

(例) 「はい」を「いいえ」に置き換えて表示します。「はい、はい」が「いいえ、はい」と表示されます。

**s** = "はい、はい"。

**ラベル** ! (s ! "はい" "いいえ" 置き換える) 作る。

**全部置き換える** : 文字列の一部を置き換えた文字列を返します。元の文字列は変更されません。置き換えは複数回行われます。置き換える文字列の指定には正規表現を使えます。

(例) 「はい」を「いいえ」に置き換えて表示します。「はい、はい」が「いいえ、いいえ」と表示されます。

**s** = "はい、はい"。

**ラベル** ! (s ! "はい" "いいえ" 全部置き換える) 作る。

**大きい整数にする** : 多倍長整数オブジェクトを作ります。10 桁以上の値を生成する場合は、数値でなく文字列から生成してください。

(例) 値が 2 の多倍長整数オブジェクトを作り、「2<sup>70</sup>」を計算します。

**x** = "2" ! 大きい整数にする。

**ラベル** ! (x ! 70 pow) 作る。

**文字コード** : 文字列の先頭文字の文字コード (UTF-16) を返します。

(例) 文字列「"あ"」の文字コードを 16 進数で表示します。「3042」が表示されます。

**ラベル** ! ("あ" ! 文字コード 16 進数) 作る。

## 真偽値

- あらかじめ「真 (はい)」「偽 (いいえ)」という 2 個のオブジェクトが用意されています。
- 論理積 (AND) や論理和 (OR) を求めるときは、それぞれ「ぜんぶ」「どれか」オブジェクトに、真偽値をパラメータとして「本当」を送ります。パラメータがブロックの場合には、その値が必要になるまでブロックは実行されません\*2。
- 論理否定 (NOT) は、真偽値に「反対」を送ります。

**本当** : 「ぜんぶ」「どれか」と組み合わせて、論理積 (AND) と論理和 (OR) を求める。パ

\*2 「ぜんぶ」「どれか」の実体は、それぞれ「はい」「いいえ」です。ブロックのパラメータが実行されるタイミングは D.4 節を参照してください。

ラメータに複数の真偽値を指定できます。

「ぜんぶ」の場合はパラメータのすべてが真のときに真を返します。

(例) 変数「x」、「y」は両方とも真（はい）なので、「"全部本当"」が表示されます。

**x=はい。y=はい。**

**「ぜんぶ！ (x) (y) 本当」！ なら「ラベル！ "全部本当" 作る」実行。**

「どれか」の場合はパラメータのすべてが偽のときに偽を返します。

(例) 変数「x」、「y」のどちらかは真（はい）なので、「"どれか本当"」が表示されます。

**x=はい。y=いいえ。**

**「どれか！ (x) (y) 本当」！ なら「ラベル！ "どれか本当" 作る」実行。**

**反対:** 真偽値と反対の値を返します。真偽値が「はい」なら「いいえ」が返り、真偽値が「いいえ」なら「はい」が返ります。

(例) 「x」の反対は真（はい）なので、「"いいえ"」を表示します。

**x=いいえ。**

**「x！ 反対」！ なら「ラベル！ "いいえ" 作る」実行。**

## ブロック

- 内部にプログラムコードを持つオブジェクトです。
- メソッド定義に用いられるほか、タイマーや繰り返しなどで利用します。
- 先頭の「|...|」でパラメータを受け取れます。「;」からはローカル変数です。
- 実行された場合は、最後に実行された値が返ります。

**繰り返す:** ブロックの中を n 回繰り返して実行します。

(例) ブロック「出力！ "こんにちは" 書く」を 3 回繰り返して実行し、「こんにちは」が 3 回表示されます。

**出力=リスト！ 作る。**

**「出力！ "こんにちは" 書く」！ 3回 繰り返す。**

何回目の実行かはパラメータとして渡されます。

(例) 実行回数をパラメータ「n」で受け取り、実行するたびに表示します。

**出力=リスト！ 作る。**

**「|n| 出力！ (n) 書く」！ 5回 繰り返す。**

**なら、そうでなければ:** 条件判断を行います。条件が成り立つときは「なら」の後のブロックを、成り立たないときは「そうでなければ」の後のブロックを実行します。「そうでなければ」は省略可能です。

(例) 乱数の値が 5 より大きい場合に「大吉」を表示します。

「乱数 (10) > 5」！ なら「ラベル！ "大吉" 作る」実行。

(例) 乱数の値が 5 より大きい場合に「大吉」を、そうでない場合は「小吉」を表示します。

「乱数 (10) > 5」！ なら「ラベル！ "大吉" 作る」そうでなければ「ラベル！ "小吉" 作る」実行。

**の間：**条件が成り立つ間、後のブロックを繰り返し実行します。

(例) 変数「x」が 10 以下の間、「s=s+x。 x=x+1」を繰り返し実行します。

x=1。 s=0。

「x <= 10」！ の間「s=s+x。 x=x+1」実行。

ラベル！ (s) 作る。

**実行：**ブロックに入っているプログラムを実行します。

(例) ブロック「ラベル！ "こんにちは" 作る」を実行します。

「ラベル！ "こんにちは" 作る」！ 実行。

## タイマー

- 一定時間ごとに、与えられたブロックを繰り返して実行します。
- 標準では、「0.1 秒」間隔で「100 回」繰り返します（約 10 秒間です）。間隔は**間隔**で変更できます。
- 回数を**回数**で指定した場合には、指定された回数を実行すると終了します。実行のパラメータで回数を指定することもできます。
- 「回数」の代りに**時間**を指定した場合は、指定された時間で実行を終了します。
- 間隔の最小時間は 1 ミリ秒 (0.001 秒) です。
- 実行される間隔は、あまり正確ではありません。たとえば 0.1 秒間隔で 10 回繰り返して実行した場合、正確に 1 秒間にはなりません。大まかな目安として使ってください。また、指定した間隔より長い時間がかかる命令を実行した場合には、「回数」を指定した実行には時間がかかってもその回数を実行し、「時間」を指定した場合にはその時間が経過した時点で繰り返しを終了します。
- タイマーはプログラムの流れと並行して（スレッドとして非同期に）実行され、プログラムはタイマーの終了を待たずに先に進みます。タイマーの終了を待つには「**待つ**」を使います。
- 実行されるブロックには、何回目の実行かを表す数がパラメータとして渡されます。
- タイマーの実行中に、そのタイマーに実行を行うと、現在の実行が終った後に続けて実行されます。
- 実行は**中断**で止めることができます。タイマーは次の実行に移ります。



- **停止**でそのタイマーの実行を完全に止めることができます。

**作る**：新しいタイマーを作ります。

(例) タイマーを作り「時計」という名前にします。

**時計=タイマー！ 作る。**

**間隔**：n 秒間隔で動くようにします。

(例) 繰り返す間隔を「1 秒」に設定します。

**時計=タイマー！ 作る。**

**時計！ 1秒 間隔。**

**回数**：n 回動くようにします。

(例) 繰り返す回数を「10 回」に設定します。

**時計=タイマー！ 作る。**

**時計！ 10回 回数。**

**時間**：n 秒間だけ動くようにします。

(例) 繰り返す時間を「5 秒」に設定します。

**時計=タイマー！ 作る。**

**時計！ 5秒 時間。**

**実行**：ブロックを実行します。

(例) タイマーを作り、「カメ太！ 3歩 歩く」を繰り返し実行します。

**カメ太=タートル！ 作る。**

**時計=タイマー！ 作る。**

**時計！ 「カメ太！ 3歩 歩く」 実行。**

回数を指定すると、その回数だけ実行します。

(例) 5 回繰り返して実行します。

**カメ太=タートル！ 作る。**

**時計=タイマー！ 作る。**

**時計！ 「カメ太！ 3歩 歩く」 5回 実行。**

何回目の実行かはパラメータとして渡されます。

(例) 何回目の繰り返しかを表示しながら実行します。

**カメ太=タートル！ 作る。**

**カウント=ラベル！ 作る。**

**時計=タイマー！ 作る。**

**時計！ 「|n| カウント！ (n) 書く。カメ太！ 3歩 歩く」 実行。**

**待つ**：タイマーが終るのを待ちます。

(例) タイマーの実行が終るのを待ってから「終了！」を表示します。「時計！ 待つ。」がない場合には、タイマーの実行中に表示されてしまいます。

**カメ太=タートル！ 作る。**

**出力=ラベル！ 作る。**

**時計=タイマー！ 作る。**

**時計！ 「|n| 出力！ (n) 書く。カメ太！ 3歩 歩く」 実行。**

**時計！ 待つ。**

**出力！ "終了！ " 書く。**

**中断：**実行中のタイマーの実行を中断します。タイマーに待ち行列がある場合は、次の実行に進みます。

(例) 中断ボタンを押すとタイマーの実行を中断します。

**カメ太=タートル！ 作る。**

**中断ボタン=ボタン！ "中断" 作る。**

**中断ボタン：動作=「時計！ 中断」。**

**出力=ラベル！ 作る。**

**時計=タイマー！ 作る。**

**時計！ 「|n| 出力！ (n) 書く。カメ太！ 3歩 歩く」 実行。**

**時計！ 「|n| 出力！ (n) 書く。カメ太！ -3歩 歩く」 実行。**

**停止：**実行中のタイマーの実行を停止します。タイマーに待ち行列がある場合は、すべての実行を停止します。

(例) 停止ボタンを押すとタイマーの実行を停止します。

**カメ太=タートル！ 作る。**

**停止ボタン=ボタン！ "停止" 作る。**

**停止ボタン：動作=「時計！ 停止」。**

**出力=ラベル！ 作る。**

**時計=タイマー！ 作る。**

**時計！ 「|n| 出力！ (n) 書く。カメ太！ 3歩 歩く」 実行。**

**時計！ 「|n| 出力！ (n) 書く。カメ太！ -3歩 歩く」 実行。**

## 配列

- 中に複数のデータを入れられるオブジェクトです。
- ひとつの配列の中に異なる種類のオブジェクトを入れることができます。
- 長さをあらかじめ決める必要はありません。
- 要素の番号は 1 から始まります。最初の要素は 1 番目です。

**作る：**新しい配列を作ります。パラメータで初期値を指定することもできます。

(例) 配列を作ります。要素はありません。

**配列1 = 配列！ 作る。**

**ラベル！ (配列1) 作る。**

(例) 文字列「"a"」、「"b"」が要素の配列を作ります。

**配列1 = 配列！ "a" "b" 作る。**

**ラベル！ (配列1) 作る。**

**書く：** 配列にオブジェクトを追加します。配列の最後に追加されます。

(例) 文字列「"a"」、「"b"」が要素の配列を作り、文字列「"c"」を追加します。

**配列1 = 配列！ "a" "b" 作る。**

**配列1！ "c" 書く。**

**ラベル！ (配列1) 作る。**

**挿入：** 配列にオブジェクトを入れます。「n obj 挿入」で、n 番目の位置に obj が挿入されます。元の n 番目以降の要素は後ろにずれます。

(例) 文字列「"a"」、「"b"」が要素の配列を作り、2 番目の位置に文字列「"c"」を追加します。「[ a c b ]」が表示されます。

**配列1 = 配列！ "a" "b" 作る。**

**配列1！ 2 "c" 挿入。**

**ラベル！ (配列1) 作る。**

**上書き：** 配列のオブジェクトを上書きします。「n obj 上書き」で、n 番目の要素が obj で上書きされます。n が配列の要素数より大きいときは、配列の大きさが拡張されて値が書かれます。

(例) 文字列「"a"」、「"b"」、「"c"」が要素の配列を作り、2 番目の要素を文字列「"d"」で上書きします。「[ a d c ]」が表示されます。

**配列1 = 配列！ "a" "b" "c" 作る。**

**配列1！ 2 "d" 上書き。**

**ラベル！ (配列1) 作る。**

**読む：** 配列の要素を返します。要素を 1 からはじまる整数で指定します。

(例) 文字列「"a"」、「"b"」、「"c"」が要素の配列を作り、2 番目の要素を表示します。「b」が表示されます。

**配列1 = 配列！ "a" "b" "c" 作る。**

**ラベル！ (配列1！ 2 読む) 作る。**

**ランダムに選ぶ：** 配列の要素をランダムに返します。

(例) 文字列「"a"」、「"b"」、「"c"」が要素の配列を作り、要素をランダムに表示します。

**配列1 = 配列！ "a" "b" "c" 作る。**

**ラベル！ (配列1！ ランダムに選ぶ) 作る。**

**要素数？：** 配列の要素数を返します。

(例) 文字列「"a"」、「"b"」、「"c"」が要素の配列を作り、要素数を表示します。「3」が表示されます。

**配列1 = 配列！ "a" "b" "c" 作る。**

**ラベル！（配列1！ 要素数？）作る。**

**消す：** 配列の要素を消します。指定されたオブジェクトを配列からすべて削除します。

(例) 文字列「"a"」、「"b"」、「"c"」が要素の配列を作り、値が「"b"」の要素を削除します。「[ a c ]」が表示されます。

**配列1 = 配列！ "a" "b" "c" 作る。**

**配列1！ "b" 消す。**

**ラベル！（配列1）作る。**

**位置で消す：** 配列の要素を消します。要素の位置を指定して削除します。

(例) 文字列「"a"」、「"b"」、「"c"」が要素の配列を作り、1 番目の要素を削除します。「[ b c ]」が表示されます。

**配列1 = 配列！ "a" "b" "c" 作る。**

**配列1！ 1 位置で消す。**

**ラベル！（配列1）作る。**

**クリア：** 配列の要素をすべて消します。

(例) 文字列「"a"」、「"b"」、「"c"」が要素の配列を作り、すべての要素を削除します。「[ ]」が表示されます。

**配列1 = 配列！ "a" "b" "c" 作る。**

**配列1！ クリア。**

**ラベル！（配列1）作る。**

**それぞれ実行：** 配列の要素の数だけブロックを繰り返して実行します。パラメータには配列の要素が1 個ずつ渡されます。

(例) 文字列「"abc"」、「"d"」、「"wxyz"」が要素の配列を作り、それぞれの要素をパラメータとしてブロック「|x| 出力！（x！ 長さ？）書く」を3 回繰り返して実行します。「3」、「1」、「4」が表示されます。

**出力=リスト！ 作る。**

**配列1 = 配列！ "abc" "d" "wxyz" 作る。**

**配列1！ |x| 出力！（x！ 長さ？）書く」それぞれ実行。**

**連結：** パラメータで指定された要素を追加した配列を返します。パラメータに配列を指定した場合はその要素が追加されます。

(例) 文字列「"大阪"」が要素の配列1 と「"東京"」、「"北海道"」が要素の配列2 を連結します。「[ 大阪 東京 北海道 ]」が表示されます。

**配列1 = 配列！ "大阪" 作る。**

**配列2 = 配列！ "東京" "北海道" 作る。**

**配列3 = 配列！ (配列1) (配列2) 連結。**

**ラベル！ (配列3) 作る。**

**選ぶ:** 配列の各要素に対してブロックを実行し、結果が「真 (はい)」の要素からなる配列を返します。

(例) 文字列「"東京"」、「"北海道"」、「"三重"」、「"鹿児島"」が要素の配列を作り、それぞれの要素をパラメータとしてブロック「`| x | (x! 長さ?) == 2`」の値が真になる要素の配列を表示します。「`[ 東京 三重 ]`」が表示されます。

**配列1 = 配列！ "東京" "北海道" "三重" "鹿児島" 作る。**

**ラベル！ (配列1！ 「`| x | (x! 長さ?) == 2`」選ぶ) 作る。**

**加工:** 配列の要素に対してブロックを実行し、それらの結果を要素とする配列を返します。

(例) 1, 3, 5 が要素の配列1 を作り、それぞれの要素の値を 2 倍するプログラムが定義されたブロック「`| n | n * 2`」を実行します。「`[ 2 6 10 ]`」が表示されます。

**配列1 = 配列！ 1 3 5 作る。**

**配列2 = 配列1！ 「`lnl n * 2`」加工。**

**ラベル！ (配列2) 作る。**

**最大:** 最大の値を持つ要素を返します。値は数値以外の場合は文字列として比較します。

(例) 1, 5, 3 が要素の配列1 を作り、最大の要素を表示します。「5」が表示されます。

**配列1 = 配列！ 1 5 3 作る。**

**ラベル！ (配列1！ 最大) 作る。**

**最小:** 最小の値を持つ要素を返します。値は数値以外の場合は文字列として比較します。

(例) 1, 5, 3 が要素の配列1 を作り、最小の要素を表示します。「1」が表示されます。

**配列1 = 配列！ 1 5 3 作る。**

**ラベル！ (配列1！ 最小) 作る。**

**結合:** 配列の各要素を結合してひとつの文字列を返します。パラメータで要素の区切り文字を指定できます。

(例) 文字列「"こんにちは、"」、「"カメ太"」、「"です！"」が要素の配列を作り、それらを結合した文字列を返します。「"こんにちは、カメ太です！"」が表示されます。

**配列1 = 配列！ "こんにちは、" "カメ太" "です！" 作る。**

**ラベル！ (配列1！ 結合) 作る。**

## ルート

- ドリトルのすべてのオブジェクトの親となるオブジェクトです。
- ルートオブジェクトのプロパティは、プログラム全体から参照される変数です。
- ルートオブジェクトのプロパティに値を書くと、他のオブジェクトからその値を参照

できます。

- ルートオブジェクトのプロパティは、オブジェクトを明示した「ルート：」、またはオブジェクトを省略した「：」で指定します。

(例) ルートオブジェクトに「歩幅」というプロパティを設定します。

**ルート：歩幅 = 30。**

(例) ルートオブジェクトに「歩幅」というプロパティを設定します。

**：歩幅 = 30。**

## 未定義

- 予期しない実行が行われた場合にシステムから返されるオブジェクトです。
- 存在しない変数を参照したときは、未定義オブジェクトが返ります。
- 命令が正しくない結果になった場合にも、未定義オブジェクトが返されることがあります。
- 値が未定義オブジェクトかどうかは、**未定義** または **undef** と比較することで判別できます。

## オブジェクトファイル

- オブジェクトをファイルに保存しておくためのオブジェクトです。
- 保存したオブジェクトは、再び読み込んで使うことができます。
- 数値、文字列、配列のオブジェクトに対応しています。
- オンライン版では利用できません。

**作る:** オブジェクトファイルを作ります。パラメータとしてファイル名を指定します。

(例) 「file1.txt」というファイルを使う、「記録」というオブジェクトファイルを作ります。

**記録 = オブジェクトファイル! "file1.txt" 作る。**

**書く:** 名前を付けてオブジェクトを書き込みます。この名前はファイルから読み出すときのキーワードになります。オブジェクトの名前と違って構いません。

(例) オブジェクトファイルに「point」という名前で「30」という値を保存します。

**記録 = オブジェクトファイル! "file1.txt" 作る。**

**点数 = 30。**

**記録! "point" (点数) 書く。**

**読む:** 名前を指定してオブジェクトを読み出します。保存してあったオブジェクトが返されます。

(例) オブジェクトファイルから「point」という名前の値を取り出して表示します。

**記録=オブジェクトファイル！ "file1.txt" 作る。**

**得点=記録！ "point" 読む。**

**ラベル！（得点）作る。**

**消す：**名前を指定してオブジェクトを消します。

(例) オブジェクトファイルから「point」という名前の値を消します。

**記録=オブジェクトファイル！ "file1.txt" 作る。**

**記録！ "point" 消す。**

## テキストファイル

- ファイルに文字列を読み書きするためのオブジェクトです。
- ファイルに書くときは、1 行ずつ文字列を追加できます。配列を書き込む場合は、ファイルの内容が配列の内容で置き換わります。
- ファイルから読むときは、全体を配列に読み込みます。文字コードは OS ごとに異なります。
- オンライン版では利用できません。

**作る：**テキストファイルを作ります。パラメータとしてファイル名を指定します。

(例) 「text1.txt」というファイルを使う、「記録」というテキストファイルを作ります。

**記録=テキストファイル！ "text1.txt" 作る。**

**書く：**文字列を書き込みます。ファイルの末尾に文字列を 1 行追加します。

(例) テキストファイルに「"こんにちは。"」を追加します。

**記録=テキストファイル！ "text1.txt" 作る。**

**記録！ "こんにちは。" 書く。**

**ラベル！（記録！ 読む）作る。**

**全部書く：**配列を書き込みます。ファイル全体を配列の中身で置き換えます。

(例) 文字列「"カメ太"」、「"カメ吉"」、「"カメ子"」が要素の配列を作り、テキストファイルをそれらで置き換えます。

**記録=テキストファイル！ "text1.txt" 作る。**

**全員=配列！ "カメ太" "カメ吉" "カメ子" 作る。**

**記録！（全員）全部書く。**

**ラベル！（記録！ 読む）作る。**

**読む：**配列に読み出します。ファイル全体を読み、各行が要素になった配列が返されます。

(例) テキストファイルの全体を読み込んで表示します。

**記録=テキストファイル！ "text1.txt" 作る。**

**ラベル！（記録！読む）作る。**

## システム

- ドリトルや動かしているコンピュータの情報を利用するためのオブジェクトです。
- システムの情報を得るためのプロパティが用意されています。

（例）ドリトルのバージョンを表示します。

**ラベル！（システム：versionstr）作る。**

プロパティ	値	プロパティ	値
<b>versionstr</b>	バージョン文字列	<b>hostname</b>	ホスト名
<b>version</b>	バージョン番号	<b>displayWidth</b>	ディスプレイの幅（※2）
<b>javavendor</b>	Java のベンダー	<b>displayHeight</b>	ディスプレイの高さ（※2）
<b>javaversion</b>	Java のバージョン	<b>memory</b>	メモリ使用量
<b>osname</b>	OS 名	<b>user</b>	ユーザー名（※3）
<b>osversion</b>	OS バージョン	<b>lang</b>	言語（※3）
<b>ipaddress</b>	IP アドレス（※1）		

（※1）ipaddress は、コンピュータが複数の IP アドレスを持つ場合は任意の 1 個を返します。IP アドレスは、サーバーウィンドウにも表示されます。

（※2）displayWidth と displayHeight はディスプレイのサイズを返します。ウィンドウのサイズは E.3 の画面オブジェクトで取得してください。

（※3）user と lang はローカル版で利用可能です。

**表示ダイアログ：**ダイアログを表示します。

（例）「こんにちは」と表示します。

**システム！ "こんにちは" 表示ダイアログ。**

**確認ダイアログ：**確認するダイアログを表示します。

（例）質問を表示し、「はい/いいえ」の回答に合わせて文を表示します。

**回答=システム！ "今日は晴れですか？ " 確認ダイアログ。**

**「回答！」なら「文="天気です" そうでなければ「文="くもりです"」実行。**

**システム！（文） 表示ダイアログ。**

**入力ダイアログ：**入力ダイアログを表示します。

（例）入力ダイアログを表示し、入力された文を表示します。

**入力=システム！ "今日の天気は？ " 入力ダイアログ。**

**システム！（入力） 表示ダイアログ。**

**選択ダイアログ：**選択ダイアログを表示します。

（例）今日の天気を選択し、回答を表示します。



**候補=配列！ "晴れ" "曇り" "雨" "雪" 作る。**

**回答=システム！ "今日の天気は？ " (候補) 選択ダイアログ。**

**システム！ (回答) 表示ダイアログ。**

**使う：**初期化ファイルを実行します。プログラムごとの初期化ファイルを指定するときに使います。指定した名前に".ini"の拡張子を付けたファイルが読み込まれ、実行されます。

ただし、**startup.ini** という名前のファイルが存在する場合は、この命令に関わらず、自動的に読み込まれて実行されます。

(例) abc.ini という初期化ファイルを実行します。

**システム！ "abc" 使う。**

**実行：**外部コマンドを実行します。ローカル版で利用可能です。

戻り値は外部コマンドの標準出力の文字列です。複数行の場合は改行記号で連結されます。戻り値のプロパティ「stderr」は外部コマンドのエラー出力です。戻り値のプロパティ「retcode」は外部コマンドのリターンコードです。

(例) OS の「date」コマンドの結果を表示します。

**結果=システム！ "date" 実行。**

**ラベル！ (結果) 作る。**

**ラベル！ (結果:stderr) 作る。**

**ラベル！ (結果:retcode) 作る。**

**proxy：**HTTP 通信を行う際の Proxy を設定します。ポートを省略した場合は 8080 が使われます。

(例) Proxy を"proxy.eplang.jp"に設定します

**システム！ "proxy.eplang.jp" proxy。**

(例) Proxy を"proxy.eplang.jp"の 8080 に設定します。

**システム！ "proxy.eplang.jp" 8080 proxy。**

**サーバーポート：**サーバーオブジェクトと通信するポート番号を指定します。標準では 2020 が使われます。通常の利用では、ポート番号を変更する必要はありません。

(例) 通信ポートを 2000 に変更します。

**システム！ 2000 サーバーポート。**

**終了する：**ドリトルを終了します。パラメータに真 (はい) を指定すると、確認せずにドリトルを終了します。

(例) ドリトルを終了します。

**システム！ 終了する。**

(例) ドリトルを終了します。確認のダイアログは表示されません。

**システム！ (はい) 終了する。**

**日時？：**日時を「Thu Feb 05 18:35:05 JST 2009」の形式で返します。

(例) 日時を表示します。

**ラベル！(システム！日時？) 作る。**

**曜日？**：曜日を「木」の形式で返します。

(例) 曜日表示します。

**ラベル！(システム！曜日？) 作る。**

**年？**：年を西暦で返します。

(例) 年を表示します。

**ラベル！(システム！年？) 作る。**

**月？**：月を数字で返します。

(例) 月を表示します。

**ラベル！(システム！月？) 作る。**

**日？**：日を数字で返します。

(例) 日を表示します。

**ラベル！(システム！日？) 作る。**

**時刻？**：時刻を「18:35:05」の形式で返します。

(例) 時刻を表示します。

**ラベル！(システム！時刻？) 作る。**

**時？**：時間を数字で返します。

(例) 時間を表示します。

**ラベル！(システム！時？) 作る。**

**分？**：分を数字で返します。

(例) 分を表示します。

**ラベル！(システム！分？) 作る。**

**秒？**：秒を数字で返します。

(例) 秒を表示します。

**ラベル！(システム！秒？) 作る。**

**システム秒？**：システム秒を返します。2つの値の差分を取ることでミリ秒単位の経過時間を計算できます。

(例) 掛け算を10万回計算する時間を表示します。

**start = システム！システム秒？。**

**「s = 2 \* 3」！ 100000 繰り返す。**

**end = システム！システム秒？。**

**ラベル！(end - start) 作る。**

## シリアルポート

- 接続された機器とシリアルポートで通信するためのオブジェクトです。
- 「作る」に以下のパラメータを指定できます。括弧内は省略したときの値です。
  - バッファサイズ: 整数 (1024)
  - 通信速度: 整数 (9600)
  - データ長: 5/6/7/8 (8)
  - ストップビット: 1/2/1.5 (1)
  - パリティビット: none/odd/even/mark/space (none)
  - フロー制御: none/rtscts/xonxoff (none)

**作る:** 新しいシリアルポートを作ります。

(例) シリアルポートを作ります。

**入出力=シリアルポート！ 作る。**

(例) シリアルポートを作りバッファサイズと通信速度を設定します。

**入出力=シリアルポート！ 1024 115200 作る。**

(例) シリアルポートを作りバッファサイズ、通信速度、データ長、ストップビット、パリティビット、フロー制御を設定します。

**入出力=シリアルポート！ 1024 115200 8 1 "none" "rtscts" 作る。**

**開く:** ポートを開きます。

(例) 「COM1」のポートを開きます。

**入出力！ "COM1" 作る。**

**閉じる:** ポートを閉じます。

(例) ポートを閉じます。

**入出力！ 閉じる。**

**出力:** ポートにデータを出力します。

(例) ポートに文字列を出力します。

**入出力！ "AT" 出力。**

**存在？:** ポートから入力できるデータが存在する場合に真を返します。

(例) ポートにデータが存在する場合にデータを1バイト読みます。

**「入出力！ 存在？」！ なら「受信データ=入出力！ 1 値？」実行。**

**データ数？:** ポートから入力できるデータのバイト数を返します。

(例) ポートから入力できるデータ数を調べます。

**データ長=入出力！ データ数？。**

**値？:** ポートから n バイトのデータを入力します。

(例) ポートから 3 バイトのデータを読みます。

**受信データ=入出力！ 3 値？。**

**待つ:** 指定された秒数だけプログラムの実行を停止します。

(例) 2 秒間停止します。

**入出力！ 2 待つ。**

**portlist:** 存在するポート名を配列で返します。

(例) シリアルポート名の一覧を「ポート配列」という配列に格納します。

**ポート配列=シリアルポート！ portlist。**

## E.2 ネットワークオブジェクト

### サーバー

- ドリトルから起動したサーバーとオブジェクトをやり取りするためのオブジェクトです。
- サーバーとドリトルの間は、特定のポート（標準では 2020）を使って通信します。そのため、一般的には教室内など同一セグメント内で利用します。
- 扱えるオブジェクトは、数値、文字列、論理値、配列などの基本オブジェクトです。タートルやボタンなど他のオブジェクトも扱えますが、機能が制限されることがあります。
- オブジェクトを書くときは、ルートを除くすべての親オブジェクトのプロパティがオブジェクトのプロパティに複製されてから書き込まれます。

**接続:** サーバーに接続する。サーバーが起動しているコンピュータを、ホスト名か IP アドレスで指定します。

(例) サーバーにホスト名で接続します。「localhost」は自分のコンピュータになります。

**サーバー！ "localhost" 接続。**

(例) サーバーに IP アドレスで接続します。

**サーバー！ "192.168.1.10" 接続。**

**書く:** 指定した名前で、サーバーにオブジェクトを書き込みます。

(例) サーバーに「kameta」という名前です「カメ太」というオブジェクトを書きます。

**サーバー！ "localhost" 接続。**

**カメ太=タートル！ 作る。**

**サーバー！ "kameta" (カメ太) 書く。**

**読む:** 指定した名前です、サーバーからオブジェクトを読み出します。読み出したオブジェク

トが返ります。

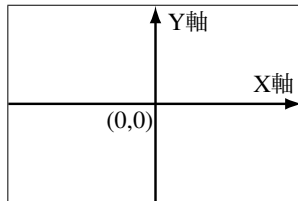
(例) サーバーから「kameta」という名前のオブジェクトを読みます。

**サーバー！ "localhost" 接続。**

**カメ吉=サーバー！ "kameta" 読む。**

## E.3 グラフィック関係のオブジェクト

次の図は**画面の座標**です。画面の中心が原点で、画面の位置は X 軸と Y 軸で表されます。座標の値は画面のピクセルと対応しています。起動したときの大きさは、横 700 × 縦 500 程度です。向きは右方向が 0 度で、左回りに指定します。



### タートル

- 画面を歩き回るキャラクタです。動くとき軌跡の線が残るので、それを利用して図形を描くことができます（タートルグラフィックス）。
- 描いた線は「図形を作る」でタートルから切り離して**図形**オブジェクトにできます。
- 「**衝突**」というメソッドを定義しておくと、他のオブジェクトと重なったときに実行されます。衝突の判定に使われるのは、タートルの本体です。描いた線を判定にしたい場合は、図形オブジェクトにしてください。
- 衝突メソッドの 1 個目のパラメータには、重なった相手のオブジェクトが渡されます。2 個目のパラメータには、自分が動いて重なったかどうか真偽値で渡されます。
- 「衝突」に「タートル：**跳ね返る**」を代入することで、自然な跳ね返りを定義できます。衝突する相手が図形オブジェクトの場合、図形オブジェクトを描き始めたときの方向が衝突する際の壁の角度として扱われます。

(例) 壁とぶつかった向きとは関係なく、常に斜め後ろを向きます。

**カメ太：衝突＝「自分！ 150 右回り」。**

(例) 壁とぶつかった向きに応じて、自然な角度で跳ね返ります。

**カメ太：衝突＝タートル：跳ね返る。**

- 「**動作**」というメソッドを定義すると、マウスでクリックしたときに実行されます。

**作る:** 新しいタートルを作ります。

(例) タートルを作り「カメ太」という名前にします。

**カメ太=タートル! 作る。**

**歩く:** 前に進みます。

(例) カメ太は 100 歩、歩きます。

**カメ太=タートル! 作る。**

**カメ太! 100歩 歩く。**

**戻る:** 後ろに戻ります。

(例) カメ太は 100 歩、戻ります。

**カメ太=タートル! 作る。**

**カメ太! 100歩 戻る。**

**右回り:** 右に回ります。

(例) カメ太は 90 度、右に回ります。

**カメ太=タートル! 作る。**

**カメ太! 90度 右回り。**

**左回り:** 左に回ります。

(例) カメ太は 90 度、左に回ります。

**カメ太=タートル! 作る。**

**カメ太! 90度 左回り。**

**移動する:** 右に x 歩、上に y 歩動きます。

(例) カメ太は「右に 0 歩、上に 100 歩」動きます。

**カメ太=タートル! 作る。**

**カメ太! 0 100 移動する。**

**位置:** 指定された座標に移動します。画面の中央が中心 (0,0) です。

(例) カメ太は「(100, 100)」の座標の位置に動きます。

**カメ太=タートル! 作る。**

**カメ太! 100 100 位置。**

**向き:** 向きを指定します。右向きが 0 度です。角度は左回りに大きくなります。

(例) カメ太は 90 度（上方向）を向きます。

**カメ太=タートル! 作る。**

**カメ太! 90度 向き。**

**ペンなし:** 動くときに軌跡が残らないようにします。

(例) カメ太は線を描かずに歩きます。

**カメ太=タートル! 作る。**

**カメ太! ペンなし。**

**カメ太! 100歩 歩く。**

**ペンあり**：動くときに軌跡が残るようにします。<sup>\*3</sup>

(例) カメ太は線を描かずに歩いた後、線を描きながら歩きます。

**カメ太=タートル！ 作る。**

**カメ太！ ペンなし。**

**カメ太！ 100歩 歩く。**

**カメ太！ ペンあり。**

**カメ太！ 100歩 歩く。**

**中心に戻る**：画面の真ん中に戻ります。

(例) カメ太は画面の中心（「(0, 0)」の座標の位置）に動きます。

**カメ太=タートル！ 作る。**

**カメ太！ 100歩 歩く。**

**カメ太！ 中心に戻る。**

**閉じる**：描き始めの点まで線を引きます。

(例) カメ太は三角形の2つの辺を描いた後、描き始めた点まで線を引いて戻ります。

**カメ太=タートル！ 作る。**

**カメ太！ 100 歩く 120 右回り 100 歩く 閉じる。**

**図形を作る**：軌跡の線を自分から切り離して図形オブジェクトにします。

(例) カメ太が描いた線を図形オブジェクトにして「三角」という名前にします。

**カメ太=タートル！ 作る。**

**カメ太！ 100 歩く 120 右回り 100 歩く。**

**三角=カメ太！ 図形を作る。**

色を指定するとその色で塗られます。

(例) カメ太が描いた線を黄色の色を塗った図形オブジェクトにして「三角」という名前にします。

**カメ太=タートル！ 作る。**

**カメ太！ 100 歩く 120 右回り 100 歩く。**

**三角=カメ太！（黄）図形を作る。**

**変身する**：タートルの姿を変えます。あらかじめ利用できる画像は、ドリトルのダウンロードページで確認してください。独自の画像を利用する場合は、画像ファイル（png, jpg, gif のいずれか）をドリトル本体（dolittle.jar）と同じディレクトリに置いてください。

(例) カメ太の姿を「tulip.png」という画像に変更します。

**カメ太=タートル！ 作る。**

<sup>\*3</sup> V2.1 から、「ペンあり」を実行しても、それまでに描いた線が図形として切り離されないようになりました。V2.0 までのプログラムで「ペンあり」で線を切り離していた場合は、必要に応じてプログラムを修正してください。

**カメ太！ "tulip.png" 変身する。**

画像を URL で指定します。

(例) カメ太の姿を「<http://dolittle.eplang.jp/image/pukiwiki.png>」の画像に変更します。

**カメ太=タートル！ 作る。**

**カメ太！ "<http://dolittle.eplang.jp/image/pukiwiki.png>" 変身する。**

画像を Twitter のアイコンで指定します。<sup>\*4</sup>

(例) カメ太の姿を「@watayan」というユーザーの画像に変更します。

**カメ太=タートル！ 作る。**

**カメ太！ "@watayan" 変身する。**

**拡大する：** タートルを拡大します。「n 拡大する」で n 倍に拡大します。n は正の整数です。

(例) カメ太を「縦横 2 倍」に拡大します。

**カメ太=タートル！ 作る。**

**カメ太！ 2 拡大する。**

「m n 拡大する」で、「横に m 倍、縦に n 倍」に拡大します。m, n は正の整数です。

(例) カメ太を横に 3 倍、縦に 2 倍に拡大します。

**カメ太=タートル！ 作る。**

**カメ太！ 3 2 拡大する。**

**線の色：** 描く線の色を変えます。はじめの色は黒です。

(例) カメ太が描く線の色を「緑」に設定します。

**カメ太=タートル！ 作る。**

**カメ太！ (緑) 線の色 100 歩く。**

**線の太さ：** 描く線の太さを変えます。はじめの太さは 3 です。

(例) カメ太が描く線の太さを「5」に設定します。

**カメ太=タートル！ 作る。**

**カメ太！ 5 線の太さ。**

**カメ太！ 100 歩く。**

**消える：** 姿を消します。描いている線も消えます。

(例) カメ太を画面から消します。

**カメ太=タートル！ 作る。**

**カメ太！ 消える。**

**現れる：** 画面に現れます。消えた姿を戻すときに使います。

(例) 消えているカメ太を表示します。「消える」の例に続いて実行してください。

---

<sup>\*4</sup> 風柳メモの「Twitter アイコン URL 取得 API」を利用させていただいています。

<http://d.hatena.ne.jp/furyu-tei/20130730/1375178609>



**カメ太！ 現れる。**

**手前に表示：**他のタートルや図形オブジェクトより手前に表示します。

(例) カメ太を他のオブジェクトより手前に表示します。

**カメ太！ 手前に表示。**

**円：**指定した半径の円を描きます\*5。大きさが正の場合はタートルの右側に、負の場合は左側に描かれます。

(例) カメ太が半径 100 の円を描きます。

**カメ太=タートル！ 作る。**

**カメ太！ 100 円。**

**角形：**正  $n$  角形を描きます。「 $m\ n$  角形」で、1 辺が長さ  $m$  の図形（正三角形、正方形、正五角形など）を描きます。図形はタートルの右側に描かれますが、辺の長さ  $m$  を負にすると左側に描かれます。

(例) カメ太が右側に 1 辺が 100 の三角形と、左側に 1 辺が 100 の五角形を描きます。

**カメ太=タートル！ 作る。**

**カメ太！ 100 3 角形。**

**カメ太！ -100 5 角形。**

**向き？：**タートルの向きを調べます。右向きが 0 度です。角度は左回りに大きくなります。

(例) 上を向いたカメ太の向きを表示します。「90」が表示されます。

**カメ太=タートル！ 作る。**

**カメ太！ 90 左回り。**

**ラベル！ (カメ太！ 向き？) 作る。**

**横の位置？：**タートルの X 座標を調べます。

(例) カメ太の X 座標を表示します。「50」が表示されます。

**カメ太=タートル！ 作る。**

**カメ太！ 60 左回り。**

**カメ太！ 100 歩く。**

**ラベル！ (カメ太！ 横の位置？) 作る。**

**縦の位置？：**タートルの Y 座標を調べます。

(例) カメ太の Y 座標を表示します。「50」が表示されます。

**カメ太=タートル！ 作る。**

**カメ太！ 30 左回り。**

**カメ太！ 100 歩く。**

**ラベル！ (カメ太！ 縦の位置？) 作る。**

---

\*5 実体は 36 角形です。

## 図形

- タートルグラフィックスで描かれた図形を独立した図形オブジェクトにできます。
- 図形を画面の上で移動したり回転させることができます。
- 回転の中心は図形を描いたときの始点です。
- 他のオブジェクトと重なると「衝突」というメソッドが実行されます。
- 「動作」というメソッドを定義すると、マウスでクリックしたときに実行されます。

**作る：**自分を複製して新しい図形を作ります。

(例) 「三角形」を複製して「三角形2」を作り、画面上で移動します。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形2 =三角形！ 作る。**

**三角形2！ 150 0 移動する。**

**右回り：**右に回ります。

(例) 三角形は 10 度、右に回ります。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 10度 右回り。**

**左回り：**左に回ります。

(例) 三角形は 10 度、左に回ります。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 10度 左回り。**

**移動する：**右に x 歩、上に y 歩動きます。

(例) 三角形は「右に 0 歩、上に 100 歩」動きます。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 0 50 移動する。**

**位置：**指定された座標に移動します。画面の中央が中心 (0,0) です。

(例) 三角形は「(100, 100)」の座標の位置に動きます。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 100 100 位置。**

**塗る：**自分の中を色で塗ります。色は色オブジェクトを括弧で囲んで書きます。色を省略す

ると線の色で塗られます。

(例) 三角形を「青」で塗ります。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ (青) 塗る。**

**拡大する：**自分を  $n$  倍します。または縦横に  $x$  倍  $y$  倍します。

(例) 三角形を「縦横 2 倍」に拡大します。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 2 拡大する。**

(例) 三角形を横に 3 倍、縦に 2 倍に拡大します。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 3 2 拡大する。**

**消える：**画面から消えます。

(例) 三角形を画面から消します。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 消える。**

**現れる：**画面に現れます。消えた姿を戻すときに使います。

(例) 消えている三角形を表示します。「消える」の例に続けて実行してください。

**三角形！ 現れる。**

**手前に表示：**他のタートルや図形オブジェクトより手前に表示します。

(例) 三角形を他のオブジェクトより手前に表示します。

**三角形！ 手前に表示。**

**向き？：**向きを調べます。右向きが 0 度です。角度は左回りに大きくなります。

(例) 三角形の向きを表示します。「90」が表示されます。

**カメ太=タートル！ 作る。**

**三角形=「カメ太！ 100 歩く 120 左回り」!3 繰り返す (赤) 図形を作る。**

**三角形！ 90 左回り。**

**ラベル！ (三角形！ 向き？) 作る。**

## 組図形

- 複数の図形をひとつのオブジェクトとして扱えます。

- 図形の**結合**メソッドで作られます。
- 要素の図形は組図形を作った後も独立した図形として扱えます。
- 回転の中心は要素の座標の中心です。
- 図形とほぼ同等のメソッドを実行できます。
- 個々の図形に定義された「衝突」イベントは要素の図形同士の衝突が発生する可能性があります。
- 以下の例では、次のように「家」という組図形を作ってから実行してください。

**かめた=タートル！ 作る。**

**屋根=「かめた！ 100 歩く 120 左回り」！ 3 繰り返す（赤）図形を作る。**

**壁=「かめた！ 100 歩く 90 右回り」！ 4 繰り返す（緑）図形を作る。**

**家=図形！（屋根）（壁）結合。**

**家！ 90 左回り。**

**作る：**自分を複製して新しい組図形を作ります。

（例）「家」を複製して「家2」を作り、画面上で移動します。

**家2 =家！ 作る。**

**家2！ 150 0 移動する。**

**右回り：**右に回ります。

（例）家は 30 度、右に回ります。

**家！ 30度 右回り。**

**左回り：**左に回ります。

（例）家は 30 度、左に回ります。

**家！ 10度 左回り。**

**移動する：**右に x 歩、上に y 歩動きます。

（例）家は「右に 0 歩、上に 100 歩」動きます。

**家！ 0 50 移動する。**

**位置：**指定された座標に移動します。画面の中央が中心（0,0）です。

（例）家は「（100, 100）」の座標の位置に動きます。

**家！ 100 100 位置。**

**塗る：**要素の図形を色で塗ります。

（例）家を「青」で塗ります。

**家！（青）塗る。**

**拡大する：**自分を n 倍します。または縦横に x 倍 y 倍します。

（例）家を「縦横 2 倍」に拡大します。

**家！ 2 拡大する。**

（例）家を横に 3 倍、縦に 2 倍に拡大します。

**家！ 3 2 拡大する。**

**消える**：画面から消えます。

(例) 家を画面から消します。

**家！ 消える。**

**現れる**：画面に現れます。消えた姿を戻すときに使います。

(例) 消えている家を表示します。「消える」の例に続けて実行してください。

**家！ 現れる。**

**手前に表示**：他のタートルや図形オブジェクトより手前に表示します。

(例) 家を他のオブジェクトより手前に表示します。

**家！ 手前に表示。**

**向き？**：向きを調べます。右向きが0度です。角度は左回りに大きくなります。

(例) 家の向きを表示します。「90」が表示されます。

**家！ 90 左回り。**

**ラベル！ (家！ 向き？) 作る。**

**画面**

- 実行画面を表すオブジェクトです。
- **マウス**という名前でも参照できます。

**塗る**：画面の背景に色を塗ります。

(例) 画面の背景を「水色」にします。

**画面！ (水色) 塗る。**

**背景画像**：画面の背景に画像を表示します。

(例) 画面の背景に「a.png」という画像を表示します。

**画面！ "a.png" 背景画像。**

**方眼紙**：画面に方眼紙の罫線を表示します。色を指定すると、その色の罫線が描かれます。

色を指定しないと罫線が消えます。

(例) 画面に「緑」の罫線を表示します。

**画面！ (緑) 方眼紙。**

(例) 画面から罫線を消します。

**画面！ 方眼紙。**

**幅？**：画面の幅を返します。

(例) 画面の横幅を表示します。

**ラベル！ (画面！ 幅？) 作る。**

**高さ？**：画面の高さを返します。

(例) 画面の高さを表示します。

**ラベル！（画面！ 高さ？）作る。**

**横の位置？**：マウスカーソルの X 座標を返します。

(例) マウスカーソルの X 座標を表示します。

**ラベル！（マウス！ 横の位置？）作る。**

**縦の位置？**：マウスカーソルの Y 座標を返します。

(例) マウスカーソルの Y 座標を表示します。

**ラベル！（マウス！ 縦の位置？）作る。**

## 色

- 色を表すオブジェクトです。
- よく使う 8 色は「黒、赤、緑、青、黄色、紫、水色、白」という変数で用意されています。
- 複数の色を混ぜ合わせるときは、パレットオブジェクトを使います。

**作る**：三原色を指定して色を作ります。赤緑青の順に 0～255 の値を指定します。

(例) 「赤が 255、緑が 136、青が 255」の明るさの色を作ります。

**ピンク＝色！ 255 136 255 作る。**

**画面！（ピンク）方眼紙。**

色を 16 進の数値で指定することもできます。赤、緑、青の明るさを、それぞれ 2 桁で指定します。

(例) 「赤が 0xFF、緑が 0x88、青が 0xFF」の色を作ります。

**ピンク＝色！ 0xFF88FF 作る。**

**画面！（ピンク）方眼紙。**

**ランダムに作る**：色をランダムに作ります。

(例) 色をランダムに作ります。実行するたびに違う色で表示されます。

**新しい色＝色！ ランダムに作る。**

**画面！（新しい色）方眼紙。**

**暗くする**：色を暗くします。

(例) 暗い緑色を作ります。

**濃い緑＝緑！ 暗くする。**

**画面！（濃い緑）方眼紙。**

**明るくする**：色を明るくします。暗くした色を再び明るくします。

(例) 暗くした色を明るくします。「暗くする」の例に続いて実行してください。

**明るい緑＝濃い緑！ 明るくする。**

**画面！（明るい緑）方眼紙。**

**半透明にする：**色を半透明にします。裏側が透けて見える半透明の色を作ります。

（例）青い半透明の色を作ります。

**新しい色＝青！ 半透明にする。**

**画面！（水色）塗る。**

**あぶく＝タートル！ 作る 30 円（新しい色）図形を作る。**

**タイマー！ 作る「あぶく！ 0.2 移動する」実行。**

## パレット

- 色を混ぜ合わせるオブジェクトです。
- あらかじめ「光」「絵具」という 2 個のオブジェクトが用意されています。
- 「光」は、光を重ねたときの色（**加法混色**）を作ります。
- 「絵具」は、絵の具を重ねたときの色（**減法混色**）を作ります。

**混ぜる：**複数の色を混ぜます。

（例）「赤」と「緑」の光を混ぜた色を作ります。

**新しい色＝光！（赤）（緑）混ぜる。**

**画面！（新しい色）塗る。**

（例）「赤」と「緑」の絵具を混ぜた色を作ります。

**新しい色＝絵具！（赤）（緑）混ぜる。**

**画面！（新しい色）塗る。**

## E.4 GUI オブジェクト

- 画面に情報を表示したり、プログラムを対話的に操作するためのオブジェクトです。
- プログラムでは、**大きさ**と**位置**で GUI オブジェクトの大きさと位置を指定して使ってください。
- 位置を指定しない場合には、直前に作った GUI オブジェクトの右隣に配置され、画面の右端を越えたり**次の行**を実行することで下の行の左端に移動します。ただし、画面の大きさは実行する環境によって異なりますので、できるだけ「大きさ」と「位置」を指定してください。
- GUI オブジェクトの背景色（**塗る**）は **Macintosh** では対応していません。
- ラベル、ボタン、リスト、選択メニューの文字列中に **HTML** を記述できます。
  - 記述できる **HTML** は、W3C の **HTML3.2** 相当です。ただし、a タグによるリンクなど、一部の機能は利用できません。**HTML** の意味や文法については、関連す

る書籍などを参照してください。

- 文字列の先頭は"<html>"で始める必要があります。""による引用は記述できません。<font color="red">は、<font color=red>と記述します。

- 利用できる主なタグを示します。

本文 (body)、見出し (h1-h6)、段落 (p, br, hr)、箇条書き (ul, ol, dl, lib)、表組 (table, tr, th, td)、文字 (b, font)、画像 (img)

- 利用例は付属のサンプルプログラム (html.dtl) をご覧ください。

- HTML を用いた画像は、タートルや図形オブジェクトと重なっても衝突を起こしません。そのため、背景画像の表示に適しています。img タグのほか、body タグの background 属性で繰り返し表示することも可能です。

(例) ローカルの large.jpg という画像を表示します。

**ラベル！ "<html><img src=file:large.jpg></html>" 作る。**

(例) 文字の背景に画像を表示します。

**ラベル！ "<html><body background=http://dolittle.eplang.jp/image/pukiwiki.png>**

**あいうえおかきくけこさしすせそ<br><br><br><br></body></html>" 作る。**

- 以下に、GUI オブジェクトに共通の命令を示します。「ボタン 1」という名前のボタンオブジェクトの例を示しています。次の 1 行に続けて書いて実行してください。

**ボタン 1 = ボタン！ "ABC" 作る。**

**次の行：**直前に作成した GUI オブジェクトの次の行に配置します。

(例) オブジェクトを直前の GUI オブジェクトの次の行に表示します。

**ボタン 1！ 次の行。**

**位置：**表示位置を指定します。

(例) オブジェクトを画面の「(100, 100)」の座標の位置に動かします。

**ボタン 1！ 100 100 位置。**

**移動する：**右に x 歩、上に y 歩動きます。

(例) オブジェクトを画面で「右に 0、上に 100」だけ動かします。

**ボタン 1！ 0 100 移動する。**

**大きさ：**大きさを指定します。

(例) オブジェクトを「横 100、縦 50」の大きさにします。

**ボタン 1！ 100 50 大きさ。**

**幅？：**横幅を調べます。

(例) オブジェクトの横幅を表示します。

**ラベル！ (ボタン 1！ 幅？) 作る。**

**高さ？：**高さを調べます。

(例) オブジェクトの高さを表示します。



**ラベル！ (ボタン1！ 高さ?) 作る。**

**文字サイズ：**表示する文字の大きさを指定します。標準は 24 ポイントです。リストとスライダー以外の GUI オブジェクトに共通です。

(例) オブジェクトに表示する文字サイズを「16」に設定します。

**ボタン1！ 16 文字サイズ。**

**塗る：**色を色オブジェクトか三原色で指定します。リスト以外の GUI オブジェクトに共通です。

(例) オブジェクトの色を設定します。

**ボタン1！ (水) 塗る。**

(例) オブジェクトの色を設定します。

**ボタン1！ 255 128 255 塗る。**

(例) オブジェクトの色を設定します。

**ボタン1！ "#FF88FF" 塗る。**

(例) オブジェクトの色を設定します。

**ボタン1！ "#F8F" 塗る。**

**文字色：**文字の色を色オブジェクトか三原色で指定します。リスト以外の GUI オブジェクトに共通です。

(例) オブジェクトの文字色を設定します。

**ボタン1！ (緑) 文字色。**

(例) オブジェクトの文字色を設定します。

**ボタン1！ 0 128 0 文字色。**

(例) オブジェクトの文字色を設定します。

**ボタン1！ "#008800" 文字色。**

(例) オブジェクトの文字色を設定します。

**ボタン1！ "#080" 文字色。**

**消える：**画面から消えます。

(例) オブジェクトを画面から消します。

**ボタン1！ 消える。**

**現れる：**画面に現れます。消えた姿を戻すときに使います。

(例) 消えているオブジェクトを画面に表示します。「消える」の例に続いて実行してください。

**ボタン1！ 現れる。**

## ボタン

- 画面に表示される GUI 部品です。初期サイズは 150×45 です。
- GUI オブジェクトに共通の説明は E.4 節を参照してください。
- ボタンが押されると**動作**というメソッドが実行されます。

(例) **ボタン1 = ボタン！ "挨拶" 作る。**

**ボタン1：動作 = 「ラベル！ "こんにちは" 作る」。**

- ボタンの生成時に**ショートカットキー**を指定することができます。
- 以下の「作る」以外の例では、「ボタン1」を作ってから実行してください。

**作る：**新しいボタンを作ります。1 個目のパラメータにはボタンに表示するラベルを指定します。

(例) 「実行」と書かれたボタンを作ります。

**ボタン1 = ボタン！ "実行" 作る。**

**ボタン1：動作 = 「ラベル！ "こんにちは" 作る」。**

2 個目のパラメータにショートカットキーを指定できます。キーの文字列は E.5 節「ショートカットキー一覧」を参照してください。次の例では上向き矢印キーを押すとボタンの動作が実行されます。

(例) 上向き矢印キーを押すと実行されるボタンを作ります。

**ボタン1 = ボタン！ "実行" "UP" 作る。**

**ボタン1：動作 = 「ラベル！ "こんにちは" 作る」。**

**読む：**文字を読んで返します。

(例) ボタンの文字列を表示します。

**ラベル！ (ボタン1！ 読む) 作る。**

**書く：**文字を書きます。

(例) ボタンに「あいうえお」という文字を書きます。

**ボタン1！ "あいうえお" 書く。**

**増やす：**表示されている数を増やします。増やす数を省略すると 1 だけ増えます。

(例) ボタンに表示されている値を 1 増やします。

**ボタン1！ 増やす。**

(例) ボタンに表示されている値を 10 増やします。

**ボタン1！ 10 増やす。**

**減らす：**表示されている数を減らします。減らす数を省略すると 1 だけ減ります。


(例) ボタンに表示されている値を 1 減らします。

**ボタン1！ 減らす。**

(例) ボタンに表示されている値を 10 減らします。

**ボタン1! 10 減らす。**

## フィールド

- 画面に表示される GUI 部品です。文字の表示や入力に使います。初期サイズは 150×45 です。
- GUI オブジェクトに共通の説明は E.4 節を参照してください。
- **リターンキー** (Enter キー, ) が押されると**動作**というメソッドが実行され、フィールドの値がパラメータとして渡されます。
- 以下の「作る」以外の例では、「フィールド1」を作ってから実行してください。

**作る**：新しいフィールドを作ります。

(例) フィールドを作ります。パラメータに初期値を指定することもできます。

**フィールド1 = フィールド! 作る。**

**読む**：文字を読んで返します。

(例) フィールドの文字を表示します。

**ラベル! (フィールド1! 読む) 作る。**

**書く**：文字を書きます。

(例) フィールドに「あいうえお」という文字を書きます。

**フィールド1! "あいうえお" 書く。**

**クリア**：空にします。

(例) フィールドの文字を消します。

**フィールド1! クリア。**

**増やす**：表示されている数を増やします。増やす数を省略すると 1 だけ増えます。

(例) フィールドに表示されている値を 1 増やします。

**フィールド1! 増やす。**

(例) フィールドに表示されている値を 10 増やします。

**フィールド1! 10 増やす。**

**減らす**：表示されている数を減らします。減らす数を省略すると 1 だけ減ります。

(例) フィールドに表示されている値を 1 減らします。

**フィールド1! 減らす。**

(例) フィールドに表示されている値を 10 減らします。

**フィールド1! 10 減らす。**

**フォーカス**：フィールドが画面で選択された状態にします。

(例) フィールドを画面でフォーカスします。

## フィールド1！ フォーカス。

### ラベル

- 画面に表示される GUI 部品です。文字の表示に使います。中に 1 つの文字列を入れます。
- GUI オブジェクトに共通の説明は E.4 節を参照してください。
- 大きさは、表示する文字列によって自動的に設定されます。
- 以下の「作る」以外の例では、「ラベル1」を作ってから実行してください。

**作る：**新しいラベルを作ります。

(例) 「あいうえお」と書かれたラベルを作ります。

**ラベル1 =ラベル！ "あいうえお" 作る。**

**書く：**文字を書きます。

(例) ラベルに「かきくけこ」という文字を書きます。

**ラベル1！ "かきくけこ" 書く。**

**増やす：**表示されている数を増やします。増やす数を省略すると 1 だけ増えます。

(例) ラベルに表示されている値を 1 増やします。

**ラベル1！ 増やす。**

(例) ラベルに表示されている値を 10 増やします。

**ラベル1！ 10 増やす。**

**減らす：**表示されている数を減らします。減らす数を省略すると 1 だけ減ります。

(例) ラベルに表示されている値を 1 減らします。

**ラベル1！ 減らす。**

(例) ラベルに表示されている値を 10 減らします。

**ラベル1！ 10 減らす。**

### リスト

- 画面に表示される GUI 部品です。初期サイズは 150×90 です。
- GUI オブジェクトに共通の説明は E.4 節を参照してください。
- 行単位で複数の文字列を入れます。配列を指定するとすべての要素が入ります。
- 以下の「作る」以外の例では、「リスト1」を作ってから実行してください。

**作る：**新しいリストを作ります。

(例) リストを作ります。

**リスト1 = リスト！ 作る。**

**書く：**文字を書きます。新しい行として追加されます。

(例) リストに「"あいうえお"」という文字列を追加します。

**リスト1！ "あいうえお" 書く。**

**読む：**文字を読んで返します。行を 1 から始まる整数で指定します。

(例) リストの 1 個目の要素を読みます。

**ラベル！ (リスト1！ 1 読む) 作る。**

**クリア：**空にします。

(例) リストのすべての要素を削除します。

**リスト1！ クリア。**

## 選択メニュー

- 画面に表示される GUI 部品です。選択肢を表示し、そこから選択できます。初期サイズは 150×45 です。
- GUI オブジェクトに共通の説明は E.4 節を参照してください。
- 選択肢が選ばれると**動作**というメソッドが実行され、選ばれた選択肢の文字列と何番目が選ばれたかを表す番号がパラメータとして渡されます。
- 以下の「作る」以外の例では、「メニュー1」を作ってから実行してください。

**作る：**新しい選択メニューを作ります。

(例) 「+」と「-」という選択肢で選択メニューを作ります。選択肢を選ぶと、その文字列が表示されます。

**メニュー1 = 選択メニュー！ "+" "-" 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」。**

**書く：**メニューに文字列を書きます。新しい選択肢として追加されます。

(例) 選択メニューに「+」と「-」という選択肢を追加します。

**メニュー1！ "+" "-" 書く。**

**何番目？：**選ばれている選択肢の番号を 1 から始まる整数で返します。「書く」の例に続けて実行してください。

(例) 選択メニューの 1 個目の選択肢が選ばれたときに、「カメ太！ 100 歩く」を実行します。

**「(メニュー1！ 何番目？) == 1」！ なら「カメ太！ 100 歩く」実行。**

**読む：**メニューの文字列を読みます。メニューの番号を 1 から始まる整数で指定します。

(例) メニューの 1 個目の選択肢の文字を読みます。「書く」の例に続けて実行してください。

**ラベル！ (メニュー1！ 1 読む) 作る。**

## スライダー

- 画面に表示される GUI 部品です。初期サイズは 300×45 です。
- GUI オブジェクトに共通の説明は E.4 節を参照してください。
- つまみを動かして 0～100 の範囲で値を変えられます。値が変わると**動作**というメソッドが実行され、スライダーの値がパラメータとして渡されます。
- スライダーの生成時に**ショートカットキー**を指定することができます。
- 以下の「作る」以外の例では、「スライダー1」を作ってから実行してください。

**作る：**新しいスライダーを作ります。

(例) スライダーを作ります。

**スライダー1 = スライダー！ 作る。**

値を減少/増加させるショートカットキーを指定できます。キーの文字列は E.5 節「ショートカットキー一覧」を参照してください。次の例では左右の矢印キーを押すとスライダーのバーが動きます。

(例) 左右の矢印キーで操作できるスライダーを作ります。値が画面に表示されます。

**スライダー1 = スライダー！ "LEFT" "RIGHT" 作る。**

**ラベル1 = ラベル！ 作る。**

**スライダー1：動作 = 「|x| ラベル1！ (x) 書く」。**

**値？：**スライダーの値を得ます。0～100 の値が返ります。

(例) スライダーの値を表示します。

**ラベル1 = ラベル！ 作る。**

**スライダー1：動作 = 「ラベル1！ (スライダー1！ 値？) 書く」。**

**値：**スライダーに値を設定します。0～100 の値を指定します。

(例) スライダーに「50」を設定します。

**スライダー1！ 50 値。**

**横向き：**横長のスライダーにします。

(例) スライダーを横向きにします。

**スライダー1！ 横向き。**

**縦向き：**縦長のスライダーにします。

(例) スライダーを縦向きにします。

**スライダー1！ 縦向き。**

**文字出す**：目盛ラベルを表示します。10 ごとの目盛ラベルが表示されます。

(例) スライダーに目盛ラベルを表示します。

**スライダー1！ 文字出す。**

**文字消す**：目盛ラベルを表示しません。

(例) スライダーから目盛ラベルを消します。

**スライダー1！ 文字消す。**

**から**：値の範囲を設定します。

**まで**：値の範囲を設定します。

(例) スライダーの値の範囲を-50 から 50 に設定します。

**スライダー1！ -50 から 50 まで。**

## E.5 ショートカットキー一覧

- ボタンやスライダーなどの GUI 部品をキーボードから操作するためのショートカットキーの一覧です。
- これらの文字は、各種のキーボードで共通に使えます。

意味	記号
英字	"A", "B", ..., "Z"
数字	"1", "2", ..., "0"
ファンクション	"F1", "F2", ..., "F12"
エスケープ (ESC)	"ESCAPE"
マイナス (−)	"MINUS"
バックスラッシュ (¥)	"BACK_SLASH"
開き括弧 ([)	"OPEN_BRACKET"
閉じ括弧 (])	"CLOSE_BRACKET"
セミコロン (;)	"SEMICOLON"
コンマ (,)	"COMMA"
ピリオド (.)	"PERIOD"
スラッシュ (/)	"SLASH"
エンター (ENTER)	"ENTER"
ホーム (HOME)	"HOME"
エンド (END)	"END"
ページアップ (PageUp)	"PAGE_UP"
ページダウン (PageDown)	"PAGE_DOWN"
カーソル上 (↑)	"UP"
カーソル下 (↓)	"DOWN"
カーソル左 (←)	"LEFT"
カーソル右 (→)	"RIGHT"

## E.6 音楽オブジェクト

- 音楽はプログラムの流れと並行して（スレッドとして非同期に）演奏され、プログラムは演奏の終了を待たずに先に進みます。演奏の終了を待つには「待つ」を使います。
- 演奏は内蔵された MIDI 音源を使って演奏されます。外部の MIDI 音源などが存在する場合は、編集画面の下部に MIDI ボタンが表示され、演奏時に使用される機器を選択できます。

### メロディ

- 音階のある旋律を表します。"ドレミ〜"のように、分かりやすい文字列でメロディを表現します。
- 以下の「作る」以外の例では、「メロディ1」などを作ってから実行してください。



**作る：**新しいメロディを作ります。

(例)「メロディ1」という名前のメロディオブジェクトを作ります。

**メロディ1 = メロディ！ 作る。**

(例)「メロディ1」という名前で「"ドレミ〜"」という音符のメロディオブジェクトを作ります。

**メロディ1 = メロディ！ "ドレミ〜" 作る。**

**設定：**楽器を設定します。

(例) メロディ1に「オルガン」の楽器オブジェクトを設定します。

**メロディ1！ (楽器！ "オルガン" 作る) 設定。**

**追加：**音符を追加します。音符は文字列とメロディオブジェクトで指定できます。

(例) メロディ1に「"ドレミ〜"」という音符を追加します。

**メロディ1！ "ドレミ〜" 追加。**

(例) メロディ2にメロディ1の音符を追加します。

**メロディ2！ (メロディ1) 追加。**

**無音：**休符を追加します。

(例) メロディ2に4拍の休符とメロディ1を追加します。

**メロディ2！ 4 無音 (メロディ1) 追加。**

**繰り返す：**繰り返したメロディを返します。

(例) メロディ2にメロディ1を2回繰り返した音符を追加します。

**メロディ2！ (メロディ1！ 2 繰り返す) 追加。**

**音上げる：**メロディを半音の個数分上げた音階で返します。

(例) メロディ1の音程を1オクターブ(半音12個分)上げたメロディ2を作ります。

**メロディ2 = メロディ1！ 12 音上げる。**

(例) メロディ1の音程を1オクターブ(半音12個分)下げたメロディ2を作ります。

**メロディ2 = メロディ1！ -12 音上げる。**

**演奏：**メロディを演奏します。

(例) メロディ1を演奏します。

**メロディ1！ 演奏。**

**待つ：**演奏が終るのを待ちます。

(例) メロディ1の演奏が終るのを待ちます。

**メロディ1！ 待つ。**

**クリア：**メロディの音符をすべて消します。

(例) メロディ1の音符をすべて消します。

**メロディ1！ クリア。**

- メロディを示す文字列には、次の表記を使えます。

ド, レ, ミ, ファ, フ, ソ, ラ, シ, ド, レ, ミ, ふぁ, ふ, そ, ら, し, C, D, E, F, G, A, B: 音階を表します。

(例) **メロディ1! "ドレミ" 追加。**

#: 半音上げます。直前の音階を半音上げます。

(例) **メロディ1! "ド#レミ" 追加。**

b (%): 半音下げます。直前の音階を半音下げます。

(例) **メロディ1! "ドレミb" 追加。**

↑, ^: **オクターブ**上げます。これ以降の音階が1オクターブ上がります。

(例) メロディ1に「"ドレミファソラシ^ドレミ"」を追加します。最後の「"ドレミ"」はオクターブ上の音階です。

**メロディ1! "ドレミファソラシ^ドレミ" 追加。**

↓, \_: **オクターブ**下げます。これ以降の音階が1オクターブ下がります。

(例) メロディ1に「"ド\_シラソファミレド"」を追加します。最後の「"ド"」以降はオクターブ下の音階です。

**メロディ1! "ド\_シラソファミレド" 追加。**

・: **休符**

(例) **メロディ1! "ド・レ・ミ" 追加。**

~ (-): **長音**。前の音を1拍延ばします。

(例) **メロディ1! "ドレミ~" 追加。**

.: 付点。前の音をその半分の長さだけ伸ばします。

(例) **メロディ1! "ドレミ." 追加。**

&, 1, 2, 4, 8, 16: 長さを指定します。「ド4レ8ミ4&8」と書くと、ドは**4分音符**、レは**8分音符**、ミは**付点4分音符**になります。

(例) **メロディ1! "ド4レ8ミ4&8" 追加。**

{...}: **三連符**。3個の音を2拍で演奏します。全体の長さを指定できます。

(例) **メロディ1! "{ドレミ}8" 追加。**

## コード

- **和音**を表します。「CCD~」のような文字列で記述します。
- 以下の「作る」以外の例では、「コード1」などを作ってから実行してください。

**作る**: 新しいコードを作ります。

(例) 「コード1」という名前のコードオブジェクトを作ります。

**コード1=コード! 作る。**

(例) 「コード1」という名前で「"CCD"」という音符のコードオブジェクトを作りま

す。

**コード1 =コード！ "CCD" 作る。**

**設定：**楽器を設定します。

(例) コード1に「オルガン」の楽器オブジェクトを設定します。

**コード1！ (楽器！ "オルガン" 作る) 設定。**

**追加：**音符を追加します。音符は文字列とコードオブジェクトで指定できます。

(例) コード1に「"CCD～"」という音符を追加します。

**コード1！ "CCD～" 追加。**

(例) コード2にコード1の音符を追加します。

**コード2！ (コード1) 追加。**

**無音：**休符を追加します。

(例) コード2に4拍の休符とコード1を追加します。

**コード2！ 4 無音 (コード1) 追加。**

**繰り返す：**繰り返したコードを返します。

(例) コード2にコード1を2回繰り返したコードを追加します。

**コード2！ (コード1！ 2 繰り返す) 追加。**

**音上げる：**コードを半音の個数分上げた音階で返します。

(例) コード1の音程を1オクターブ(半音12個分)上げたコード2を作ります。

**コード2 =コード1！ 12 音上げる。**

(例) コード1の音程を1オクターブ(半音12個分)下げたコード2を作ります。

**コード2 =コード1！ -12 音上げる。**

**演奏：**コードを演奏します。

(例) コード1を演奏します。

**コード1！ 演奏。**

**待つ：**演奏が終るのを待ちます。

(例) コード1の演奏が終るのを待ちます。

**コード1！ 待つ。**

**クリア：**追加したコードをすべて消します。

(例) コード1のコードをすべて消します。

**コード1！ クリア。**

- コードを示す文字列には、次の表記を使えます。

A, B, C, D, E, F, G: コードを表します。

(例) **コード1！ "CCG" 追加。**

m, 7: コードを修飾します。**マイナーコード**を作るときに使います。

(例) **コード1！ "CCmC7Cm7" 追加。**

**#:** 半音上げます。直前の音階を半音上げます。

(例) コード1! "C # CD" 追加。

**♭ (%):** 半音下げます。直前の音階を半音下げます。

(例) コード1! "CCD ♭" 追加。

**↑, ^:** オクターブ上げます。これ以降の音階が1オクターブ上がります。

(例) コード1に「"C^CD"」を追加します。最後の「"CD"」はオクターブ上の音階です。

コード1! "C^CD" 追加。

**↓, \_:** オクターブ下げます。これ以降の音階が1オクターブ下がります。

(例) メロディ1に「"C\_CD"」を追加します。最後の「"CD"」はオクターブ下の音階です。

コード1! "C\_CD" 追加。

**・:** 休符

(例) コード1! "C・C・D" 追加。

**〜 (-):** 長音。前の音を1拍延ばします。

(例) コード1! "CCD〜" 追加。

**.**: 付点。前の音をその半分の長さだけ伸ばします。

(例) コード1! "CCD." 追加。

**&,1,2,4,8,16:** 長さを指定します。"C4D8E4&8" と書くと、Cは4分音符、Dは8分音符、Eは付点4分音符になります。

(例) コード1! "C4D8E4&8" 追加。

**{...}:** 三連符。3個の音を2拍で演奏します。全体の長さを指定できます。

(例) コード1! "{CDE}8" 追加。

## ドラム

- ドラム楽器の演奏を表します。"ドツタツ" のように、分かりやすい文字列でリズムを表現します。
- 楽器を設定せず、ドラムオブジェクトで演奏してください。単独で演奏できるほか、バンドオブジェクトのメンバーとして他の楽器などと同時に演奏できます。
- 以下の「作る」以外の例では、「ドラム1」などを作ってから実行してください。

**作る:** 新しいドラムを作ります。

(例) 「ドラム1」という名前のドラムオブジェクトを作ります。

ドラム1 = ドラム! 作る。

**追加:** 音符を追加します。音符は文字列とドラムオブジェクトで指定できます

(例) ドラム1 に「"ドツタツ"」という音符を追加します。

**ドラム1！ "ドツタツ" 追加。**

(例) ドラム2 にドラム1 の音符を追加します。

**ドラム2！（ドラム1）追加。**

**無音：** 休符を追加します。

(例) ドラム1 に 4 拍の休符を追加します。

**ドラム1！ 4 無音。**

**繰り返す：** 繰り返したドラムを返します。

(例) ドラム2 にドラム1 を 2 回繰り返した音符を追加します。

**ドラム2！（ドラム1！ 2 繰り返す）追加。**

**演奏：** ドラムを演奏します。

(例) ドラム1 を演奏します。

**ドラム1！ 演奏。**

**待つ：** 演奏が終るのを待ちます。

(例) ドラム1 の演奏が終るのを待ちます。

**ドラム1！ 待つ。**

**音量：** 音の大きさを設定します。値は 0～127 で、標準の大きさは 95 です。

(例) 「ドラム1」の音の大きさを「127」に設定します。

**ドラム1！ 127 音量。**

**クリア：** 追加したドラムをすべて消します。

(例) ドラム1 の音符をすべて消します。

**ドラム1！ クリア。**

**楽器設定：** 音符と楽器の対応を設定します。楽器は楽器名の文字列または楽器番号で指定します。複数の楽器を設定できます。

(例) ドラム1 の「"ド"」を「"手拍子"」という楽器に設定します。

**ドラム1！ "ド" "手拍子" 楽器設定。**

(例) ドラム1 の「"ド"」を「36」番の楽器に設定します。

**ドラム1！ "ド" 36 楽器設定。**

(例) ドラム1 の「"ドタツクチバ"」を、それぞれ「35, 38, 42, 44, 46, 49」番の楽器に設定します。

**ドラム1！ "ドタツクチバ" 35 38 42 44 46 49 楽器設定。**

- ドラムを示す文字列には、次の表記を使えます。楽器の割当は「楽器設定」で変更できます。

(例) **ドラム！ 作る "ドツタツドツタツドツタツクチバ・" 追加 演奏。**

ド, (ど)：バスドラの半拍を表します。

タ、(た)：スネアの半拍を表します。

ツ、(つ)：ハイハット（クローズ）の半拍を表します。

ク、(く)：ハイハット（ハーフオープン）の半拍を表します。

チ、(ち)：ハイハット（オープン）の半拍を表します。

パ、(ぱ)：クラッシュシンバルの半拍を表します。

ン（ん）：**長音**。前の拍を半拍長くします。

・：休符。1 拍休みます。

&,1,2,4,8,16: 長さを示します。"タン 4 タン 8 タン 4&8" と書くと、最初のタンは **4**

**分音符**、次のタンは **8 分音符**、最後のタンは**付点 4 分音符**になります。

{...}: **三連符**。3 個の音を 2 拍で演奏します。全体の長さを指定できます。

- ドラムの「楽器設定」で使える楽器名と楽器番号には、次のものが使えます。

番号	名前	番号	名前	番号	名前
35	バスドラム 2	51	ライドシンバル 1	67	ハイアゴゴ
36	バスドラム 1	52	チャイニーズシンバル	68	ローアゴゴ
37	サイドスティック	53	ライドベル	69	カバサ
38	スネアドラム 1	54	タンバリン	70	マラカス
39	手拍子	55	スブラッシュシンバル	71	ショートホイッスル
40	スネアドラム 2	56	カウベル	72	ロングホイッスル
41	ロートム 2	57	クラッシュシンバル 2	73	ショートギロ
42	クローズハイハット	58	ヴィブラスラップ	74	ロングギロ
43	ロートム 1	59	ライドシンバル 2	75	クラヴェス
44	ペダルハイハット	60	ハイボンゴ	76	ハイウッドブロック
45	ミドルトム 2	61	ローボンゴ	77	ローウッドブロック
46	オープンハイハット	62	ミュートハイコンガ	78	ミュートクイーカ
47	ミドルトム 1	63	オープンハイコンガ	79	オープンクイーカ
48	ハイトム 2	64	ローコンガ	80	ミュートトライアングル
49	クラッシュシンバル 1	65	ハイタンバール	81	オープントライアングル
50	ハイトム 1	66	ロータンバール		

楽器

- メロディとコードを演奏する楽器です。
- 以下の「作る」以外の例では、「ピアノ1」を作ってから実行してください。

**作る**：新しい楽器を作ります。

(例)「ピアノ1」という名前の楽器オブジェクトを作ります。

**ピアノ1 = 楽器！ "ピアノ" 作る。**

**設定**：演奏するメロディとコードを設定します。

(例)「ピアノ1」に「メロディ1」のメロディを設定します。

**ピアノ1！（メロディ1）設定。**

(例) 「ピアノ1」に「コード1」のコードを設定します。

**ピアノ1! (コード1) 設定。**

**演奏:** 楽器を演奏します。

(例) 「ピアノ1」を演奏します。

**ピアノ1! 演奏。**

**待つ:** 演奏が終るのを待ちます。

(例) 「ピアノ1」の演奏が終るのを待ちます。

**ピアノ1! 待つ。**

**音量:** 音の大きさを設定します。値は 0~127 で、標準の大きさは 95 です。

(例) 「ピアノ1」の音の大きさを「127」に設定します。

**ピアノ1! 127 音量。**

- 楽器名と楽器番号には、次のものが使えます。

番号	名前	番号	名前	番号	名前
1	グランドピアノ	44	コントラバス	87	フィフスリード
2	ブライトピアノ	45	トレモロストリングス	88	ベースアンドリード
3	エレクトリックグランドピアノ	46	ピチカートストリングス	89	ニューエイジパッド
4	ホンキートンクピアノ	47	オーケストラハーブ	90	ワームパッド
5	エレクトリックピアノ 1	48	ティンパニ	91	ポリシンセパッド
6	エレクトリックピアノ 2	49	ストリングス	92	クワイアパッド
7	ハーブシコード	50	スローストリングス	93	ボウドラッド
8	クラビネット	51	シンセストリングス 1	94	メタリックパッド
9	チェレスタ	52	シンセストリングス 2	95	ハロパッド
10	グロッケンシュピール	53	コーラス	96	スウィープパッド
11	ミュージックボックス	54	ボイス	97	アイスレイン
12	ビブラフォン	55	シンセボイス	98	サウンドトラック
13	マリンバ	56	オーケストラヒット	99	クリスタル
14	シロフォン	57	トランペット	100	アトモスフィア
15	チューブラーベル	58	トロンボーン	101	ブライトネス
16	ダルシマー	59	チューバ	102	ゴブリン
17	ドローバーオルガン	60	ミュートトランペット	103	エコードロップ
18	パーカッションオルガン	61	フレンチホルン	104	エスエフ
19	ロックオルガン	62	プラスセクション	105	シタール
20	チャーチオルガン	63	シンセプラス 1	106	バンジョー
21	リードオルガン	64	シンセプラス 2	107	三味線
22	アコーディオン	65	ソプラノサククス	108	琴
23	ハーモニカ	66	アルトサククス	109	カリンバ
24	タンゴアコーディオン	67	テナーサククス	110	バグパイプ
25	ナイロンギター	68	バリトンサククス	111	フィドル
26	スティールギター	69	オーボエ	112	シャナイ
27	ジャズギター	70	イングリッシュホルン	113	ティンクルベル
28	クリーンギター	71	バスーン	114	アゴゴ
29	ミュートギター	72	クラリネット	115	スティールドラム
30	オーバードライブギター	73	ビッコロ	116	ウッドブロック
31	ディストーションギター	74	フルート	117	太鼓
32	ギターハーモニクス	75	リコーダー	118	メロディックタム
33	アコースティックベース	76	バンフルート	119	シンセドラム
34	フィンガーベース	77	ブローボトル	120	リバースシンバル
35	ピックベース	78	尺八	121	ギターフレットノイズ
36	フレットレスベース	79	ホイッスル	122	プレスノイズ
37	スラップベース 1	80	オカリナ	123	シーショア
38	スラップベース 2	81	スクウェアリード	124	バード
39	シンセベース 1	82	ソートゥースリード	125	テレフォン
40	シンセベース 2	83	カリオペ	126	ヘリコプター
41	バイオリン	84	チフリード	127	アブローズ
42	ビオラ	85	チャランゴ	128	ガンショット
43	チェロ	86	ボイスリード		



## バンド

- 複数の楽器やメロディ/コード/ドラムを演奏します。
- 以下の「作る」以外の例では、「マイバンド」を作ってから実行してください。

**作る**：新しいバンドを作ります。

(例) 「マイバンド」という名前のバンドオブジェクトを作ります。

**マイバンド=バンド！ 作る。**

**追加**：演奏する楽器/メロディ/コード/ドラムを設定します。

(例) 「マイバンド」に「ピアノ1」という楽器を追加します。

**マイバンド！（ピアノ1）追加。**

**演奏**：バンドを演奏します。

(例) 「マイバンド」を演奏します。

**マイバンド！ 演奏。**

**待つ**：演奏が終るのを待ちます。

(例) 「マイバンド」の演奏が終るのを待ちます。

**マイバンド！ 待つ。**

**クリア**：バンドのメンバー（楽器/メロディ/コード/ドラム）をすべて取り消します。

(例) 「マイバンド」に追加された楽器/メロディ/コード/ドラムのメンバーをすべて取り消します。

**マイバンド！ クリア。**

**テンポ**：演奏する速度を指定します。標準は 88 です。

(例) 「マイバンド」の演奏する速度を「100」に設定します。

**マイバンド！ 100 テンポ。**

## E.7 Arduino オブジェクト

### Arduino

- Arduino を制御するためのオブジェクトです。
- これらのオブジェクトを使うプログラムでは、先頭に次の 1 行を記述してください。

**システム！ "arduino" 使う。**

**作る**：Arduino オブジェクトを作ります。

(例) a1 という名前の Arduino オブジェクトを作ります。

**a1 = Arduino！ 作る。**

**ひらけごま**：ポートを開きます。

(例) 「COM1」というポートを指定して開きます。

**a1 ! "COM1" ひらけごま。**

(例) ダイアログでポートを選択しながら開きます。

**a1 ! (システム! シリアルポート選択) ひらけごま。**

**とじろごま**：ポートを閉じます。

(例) 開いているポートを閉じます。

**a1 ! とじろごま。**

**待つ**：指定した秒数だけ実行を止めます。

(例) 1 秒間、実行を止めます。

**a1 ! 1 待つ。**

**デジタル出力**：デジタル出力用のオブジェクトを作ります。

(例) 13 番ポートにデジタル出力をするためのオブジェクトを作ります。

**led1 = a1 ! 13 デジタル出力。**

**デジタル入力**：デジタル入力用のオブジェクトを作ります。

(例) 3 番ポートからデジタル入力をするためのオブジェクトを作ります。

**sw1 = a1 ! 3 デジタル入力。**

**アナログ出力**：アナログ出力用のオブジェクトを作ります。

(例) 9 番ポートにアナログ出力をするためのオブジェクトを作ります。

**led2 = a1 ! 9 アナログ出力。**

**アナログ入力**：アナログ入力用のオブジェクトを作ります。

(例) 0 番ポートからアナログ入力をするためのオブジェクトを作ります。

**cds1 = a1 ! 0 アナログ入力。**

## デジタル出力

- デジタル出力を行うオブジェクトです。
- 出力する値は、0 と 1 の数値を指定できます。

**書く**：ポートにデータを出力します。

(例) ポートに 1 を出力します。

**led1 ! 1 書く。**

**待つ**：指定した秒数だけ実行を止めます。

(例) 1 秒間、実行を止めます。

**led1 ! 1 待つ。**

## デジタル入力

- デジタル入力を行うオブジェクトです。
- 入力される値は 0 か 1 の数値です。

**読む**：ポートからデータを入力します。

(例) ポートからデータを入力します。

**入力 = sw1 ! 読む。**

**待つ**：指定した秒数だけ実行を止めます。

(例) 1 秒間、実行を止めます。

**sw1 ! 1 待つ。**

## アナログ出力

- アナログ出力を行うオブジェクトです。
- 出力する値は、0 から 255 の数値を指定できます。

**書く**：ポートにデータを出力します。

(例) ポートに 255 を出力します。

**led2 ! 255 書く。**

**待つ**：指定した秒数だけ実行を止めます。

(例) 1 秒間、実行を止めます。

**led2 ! 1 待つ。**

## アナログ入力

- アナログ入力を行うオブジェクトです。
- 入力される値は 0 から 255 の数値です。

**読む**：ポートからデータを入力します。

(例) ポートからデータを入力します。

**入力 = cds1 ! 読む。**

**待つ**：指定した秒数だけ実行を止めます。

(例) 1 秒間、実行を止めます。

**cds1 ! 1 待つ。**

## E.8 LeapMotion オブジェクト

- LeapMotion の値を取得するためのオブジェクトです。
- 空中の手指の位置や動きを検出することができます。
- LeapMotion は 1 台だけ接続して使うことができます。
- 手指の動き（ジェスチャー）によって次のメソッドが実行されます。
  - **回転**: 正面から見たときの動きが円に見えるように手指を動かしたときに実行されます。回転方向が時計回りかどうかを示す真偽値がパラメーターとして渡されます。
  - **スワイプ**: 手指を直線的に動かしたときに実行されます。右方向への移動かどうかを示す真偽値がパラメーターとして渡されます。
  - **タップ**: 手指を正面に向かって突き出したときに実行されます。横方向の位置を示す数値と高さを示す数値がパラメーターとして渡されます。
  - **キータップ**: 手指を下方方向に動かしたときに実行されます。横方向の位置を示す数値がパラメーターとして渡されます。
- LeapMotion オブジェクトは「リープ」「リープモーション」「leap」「leapmotion」という名前で使用できます。
- これらのオブジェクトはあらかじめ用意されており、「作る」を実行する必要はありません。

**接続**: LeapMotion との通信を開始します。

(例) LeapMotion との通信を開始します。

**リープ！ 接続。**

**横の位置?**: 空間での手のひらの左右の座標を取得します。中央が 0 で、右は正の値、左は負の値です。

(例) 手のひらの横の座標を取得します。

**x = リープ！ 横の位置?。**

**縦の位置?**: 空間での手のひらの高さを取得します。LeapMotion 本体からの高さです。

(例) 手のひらの高さを取得します。

**y = リープ！ 縦の位置?。**

**前後の位置?**: 空間での手のひらの前後の位置を取得します。LeapMotion の真上が 0 で、手前は正の値、手を前方に伸ばすと負の値になります。

(例) 手のひらの前後の位置を取得します。

**z = リープ！ 前後の位置?。**

**指の数?**: 手の指の数を取得します。

(例) 指の数を取得します。

**n = リープ！ 指の数？。**

**グー？**：手の指の数からじゃんけんの「グー」かどうかを判断します。

(例) グーかどうかを判断します。

**「リープ！ グー？」！ なら「ラベル！ "グー" 作る」実行。**

**チョキ？**：手の指の数からじゃんけんの「チョキ」かどうかを判断します。

(例) チョキかどうかを判断します。

**「リープ！ チョキ？」！ なら「ラベル！ "チョキ" 作る」実行。**

**パー？**：手の指の数からじゃんけんの「パー」かどうかを判断します。

(例) パーかどうかを判断します。

**「リープ！ パー？」！ なら「ラベル！ "パー" 作る」実行。**

**回転？**：手のひらが縦方向に円を描いているかどうかを返します。

(例) 手のひらが円を描いていれば「回転中」と表示します。

**「リープ！ 回転？」！ なら「ラベル！ "回転中" 作る」実行。**

**左回転？**：手のひらが反時計回りで円を描いているかどうかを返します。

(例) 手のひらが反時計回りに円を描いていれば「左回転中」と表示します。

**「リープ！ 左回転？」！ なら「ラベル！ "左回転中" 作る」実行。**

**右回転？**：手のひらが時計回りで円を描いているかどうかを返します。

(例) 手のひらが時計回りに円を描いていれば「右回転中」と表示します。

**「リープ！ 右回転？」！ なら「ラベル！ "右回転中" 作る」実行。**

**スワイプ？**：手のひらが直線的に動いているかどうかを返します。

(例) 手のひらが直線的に動いていれば「移動中」と表示します。

**「リープ！ スワイプ？」！ なら「ラベル！ "移動中" 作る」実行。**

**左スワイプ？**：手のひらが左の方向に直線的に動いているかどうかを返します。

(例) 手のひらが左方向に直線的に動いていれば「左に移動中」と表示します。

**「リープ！ 左スワイプ？」！ なら「ラベル！ "左に移動中" 作る」実行。**

**右スワイプ？**：手のひらが右の方向に直線的に動いているかどうかを返します。

(例) 手のひらが右方向に直線的に動いていれば「右に移動中」と表示します。

**「リープ！ 右スワイプ？」！ なら「ラベル！ "右に移動中" 作る」実行。**



# 索引

!, 4, 145  
 !=, 147, 167, 168, 179, 181, 182  
 ≠, 179, 181, 182  
 \*, 147, 167, 168, 178, 181  
 ×, 167, 168, 178, 181  
 +, 147, 167–169, 178, 181, 183  
 −, 147, 167, 168, 178, 181  
 ∼, 220, 222  
 −, 220, 222  
 •, 220, 222  
 /, 147, 167, 168, 178, 181  
 ÷, 167, 168, 178, 181  
 ∴, 144  
 <, 147, 167, 168, 179, 181, 182  
 <=, 147, 167, 168, 179, 181, 182  
 ≤, 179, 181, 182  
 =, 3, 144  
 ==, 147, 167, 168, 179, 181, 182  
 >, 147, 167, 168, 179, 181, 182  
 >=, 147, 167, 168, 179, 181, 182  
 ≥, 179, 181, 182  
 ♭, 220, 222  
 #, 220, 222  
 %, 147, 167, 168, 178, 181, 220, 222  
 &, 220, 222  
 −, 220, 222  
 ^, 220, 222  
 ↑, 220, 222  
 ↓, 220, 222  
 4 分音符, 220, 222, 224  
 8 分音符, 220, 222, 224  
  
 abs, 167, 168, 179, 181  
 acos, 179  
 add, 178, 181  
 AND, 170, 184  
 Arduino, 68, 227  
 Arduino IDE, 69  
 arduino\_dolittle.ino, 69  
 asin, 179  
 atan, 179  
 atan2, 179  
  
 CdS, 72

ceil, 179  
 cos, 167, 179  
  
 displayHeight, 194  
 displayWidth, 194  
 div, 178, 181  
 DOWN, 35, 218  
 dq, 182  
  
 exp, 179  
  
 floor, 179  
  
 GUI 部品, 15, 19, 173  
  
 hostname, 194  
  
 ipaddress, 194  
 IP アドレス, 112  
  
 javavendor, 194  
 javaversion, 194  
  
 lang, 194  
 ldb, 182  
 ldq, 182  
 leap, 230  
 LeapMotion, 64  
 leapmotion, 230  
 LED, 70, 83  
 LEFT, 31, 40, 218  
 ln, 179  
 localhost, 112  
 log, 179  
  
 Macintosh, 209  
 Macintosh 版, iv  
 memory, 194  
 MIDI 音源, 48  
 mod, 178, 181  
 mul, 178, 181  
 MYU オブジェクト, 95  
 MYU ロボ, 92  
  
 NOT, 170, 184

OR, 170, 184  
 osname, 194  
 osversion, 194  
  
 π, 178  
 PI, 178  
 portlist, 198  
 pow, 168, 179, 181  
 proxy, 195  
 PWM, 71  
  
 random, 180  
 rdb, 182  
 rdq, 182  
 RIGHT, 31, 40, 218  
 round, 167, 179  
  
 server, 131  
 sin, 167, 179  
 sqrt, 167, 179  
 startup.ini, 195  
 Studuino, 78  
 sub, 178, 181  
  
 tan, 167, 179  
  
 undef, 122, 176, 192  
 UP, 35, 41, 218  
 user, 194  
  
 version, 194  
 versionstr, 194  
  
 Web ブラウザ, v  
  
 青, 16, 171, 208  
 赤, 16, 171, 208  
 明るくする, 16, 208  
 値, 216  
 値? , 197, 216  
 アナログ出力, 71, 228, 229  
 アナログ入力, 72, 87, 228, 229  
 余り, 178, 181  
 現れる, 202, 205, 207, 211  
 歩く, 4, 200

いいえ, 37, 170, 184  
 位置, 14, 15, 20, 200, 204, 206, 209, 210  
 位置で消す, 158, 190  
 移動する, 15, 20, 154, 200, 204, 206, 210  
 色, 16, 171, 208  
 インスタンス変数, 162  
 引用符, 182  
 上書き, 157, 189  
 絵具, 17, 171, 209  
 選ぶ, 191  
 円, 203  
 円周率, 178  
 演奏, 48, 219, 221, 223, 225, 227  
 大きい整数にする, 168, 180, 184  
 大きさ, 20, 209, 210  
 置き換える, 184  
 オクターブ, 220, 222  
 音上げる, 219, 221  
 オブジェクト, 2, 143  
 オブジェクト指向言語, 144  
 オブジェクトファイル, 175, 192  
 親子関係, 159  
 オルガン, 53  
 おわりロボット, 95, 96, 108  
 音階, 220  
 オンライン版, iii  
 音量, 223, 225  
 改行, 164  
 回数, 24, 25, 153, 186, 187  
 回転, 230  
 回転?, 231  
 書く, 113, 157, 175, 189, 192, 193, 198, 212–215, 228, 229  
 拡大する, 202, 205, 206  
 拡張 BNF, 163  
 確認ダイアログ, 194  
 掛ける, 178, 181  
 加工, 191  
 カスケード, 6, 41, 146  
 角形, 203  
 加法混色, 17, 209  
 から, 217  
 間隔, 24, 25, 153, 186, 187  
 楽器, 53, 224  
 楽器設定, 223  
 画面, 194, 207  
 画面の座標, 199  
 黄色, 16, 171, 208  
 消える, 32, 202, 205, 207, 211  
 休符, 220, 222  
 曲の構造, 50  
 キータップ, 230  
 偽, 170, 184  
 空白, 4, 9, 10, 164  
 組図形, 205  
 暗くする, 16, 208

クリア, 190, 213, 215, 219, 221, 223, 227  
 繰り返し, 7, 23, 51, 97  
 繰り返し脱出, 96, 98, 101, 108  
 繰り返し実行, 84  
 繰り返す, 96, 101, 108, 149, 185, 219, 221, 223  
 黒, 16, 171, 208  
 グローバル変数, 161  
 グー, 66  
 グー?, 231  
 消す, 158, 190, 193  
 結合, 191, 206  
 減法混色, 17, 209  
 後退, 96, 108  
 コピー&ペースト, v  
 コメント, 139, 166  
 コンピュータ名, 112  
 コード, 48, 220  
 コード文字, 180, 182  
 最小, 191  
 最初に実行, 84  
 最大, 191  
 三原色, 17  
 三連符, 220, 222, 224  
 サーバー, 198  
 サーバーポート, 195  
 サーボモーター, 89  
 識別子, 164  
 システム, 194  
 システム秒?, 196  
 終了する, 195  
 出力, 197  
 衝突, 32, 36, 199, 204  
 触覚センサー, 93  
 ショートカットキー, 31, 212, 216, 217  
 シリアルポート, 197  
 シリアルポート選択, 92  
 白, 16, 171, 208  
 真, 170, 184  
 真偽値, 170, 184  
 進数, 180  
 時間, 25, 153, 186, 187  
 字句, 163  
 時刻?, 196  
 実行, 25, 148, 150, 151, 153, 183, 186, 187, 195  
 実行画面, v  
 実行時エラー, 10  
 実行脱出, 101, 108  
 実行ボタン, v, 2  
 自分, 8, 145, 163  
 条件式, 51  
 時?, 196  
 数値, 165, 167, 168, 175, 178  
 ストップビット, 197  
 スライダー, 174, 216  
 スレッド, 26, 155

スワイプ, 230  
 スワイプ?, 231  
 図形, 14, 172, 199, 204  
 図形を作る, 14, 201  
 赤外線センサー, 88  
 赤外線フォトリフレクタ, 88  
 設定, 53, 219, 221, 224  
 接続, 113, 198, 230  
 選択ダイアログ, 194  
 選択メニュー, 49, 174, 215  
 線の色, 202  
 線の太さ, 35, 125, 202  
 旋律, 48  
 前後の位置?, 230  
 前進, 96, 108  
 ぜんぶ, 170, 184  
 全部置き換える, 184  
 全部書く, 193  
 そうでなければ, 151, 185  
 挿入, 189  
 束縛, 161  
 それぞれ実行, 158, 190  
 存在?, 197  
 タイマー, 153, 186  
 タイマーオブジェクト, 23  
 タイマーの逐次実行, 42  
 高さ?, 207, 210  
 足す, 178, 181  
 タップ, 230  
 縦の位置?, 203, 208, 230  
 縦向き, 216  
 多倍長整数, 168, 180  
 タブ, v  
 タートル, 2, 3, 171, 172, 199  
 タートルグラフィックス, 2  
 代入, 144  
 ダブルクォーテーション, 182  
 ダブルクオート, 182  
 チャット, 115  
 注釈, 166  
 中心に戻る, 201  
 中断, 186, 188  
 中置記法, 147, 178  
 長音, 220, 222, 224  
 チョキ?, 231  
 追加, 48, 219, 221, 222, 227  
 通信速度, 197  
 使う, 70, 83, 95, 195  
 月?, 196  
 次の行, 209, 210  
 作る, 2, 4, 15, 19, 159, 173, 175, 177, 187, 188, 192, 193, 197, 200, 204, 206, 208, 212–216, 219, 220, 222, 224, 227  
 停止, 96, 101, 108, 187, 188  
 テキストファイル, 177, 193  
 手前に表示, 203, 205, 207  
 転送, 84



- テンポ, 227  
 デジタル出力, 70, 228  
 デジタル入力, 71, 86, 228, 229  
 デバッグ, 10, 140  
 電子音, 101, 109  
 データ数?, 197  
 データ長, 197  
 閉じる, 197, 201  
 とじろごま, 228  
 同期, 27  
 動作, 19, 173–175, 199, 204, 212, 213, 215, 216  
 ドラム, 48, 56, 222  
 ドリトルの Web サイト, v  
 どれか, 170, 184  
 長さ?, 183  
 名前, 3, 144, 164  
 なら, 151, 185  
 何番目?, 215  
 何文字目?, 183  
 日時?, 195  
 日?, 196  
 入力あり, 96, 101, 102, 108  
 入力ダイアログ, 194  
 入力なし, 108  
 塗る, 15, 20, 204, 206, 207, 209, 211  
 年?, 196  
 の間, 101, 108, 150, 186  
 はい, 37, 170, 184  
 背景画像, 207  
 配列, 57, 157, 175, 188  
 はじめロボット, 95, 96, 108  
 発光ダイオード, 70  
 跳ね返る, 36, 199  
 幅?, 207, 210  
 半音, 220, 222  
 反対, 170, 184, 185  
 半透明にする, 209  
 半拍, 223  
 バッファサイズ, 197  
 バンド, 48, 52, 222, 227  
 パラメータ, 4, 51, 145  
 パリティビット, 197  
 パレット, 17, 171, 209  
 パワーオンスタート, 97, 108  
 パー, 66  
 パー?, 231  
 光, 17, 171, 209  
 光センサ, 72, 87  
 引く, 178, 181  
 左後, 108  
 左回転?, 231  
 左スワイプ?, 231  
 左ダブルクォーテーション, 182  
 左ダブルクォート, 182  
 左二重かぎ括弧, 182  
 左前, 108  
 左回り, 4, 15, 96, 108, 200, 204, 206  
 表示ダイアログ, 194  
 開く, 197  
 ひらけごま, 228  
 秒?, 196  
 ピアノ, 48  
 フィールド, 44, 49, 174, 213  
 フォーカス, 213  
 含む?, 169, 183  
 付点 4 分音符, 220, 222, 224  
 増やす, 212–214  
 フロー制御, 197  
 分?, 196  
 ブザー, 109  
 部分, 169, 183  
 ブレッドボード, 73  
 ブロック, 7, 8, 51, 57, 100, 102, 148, 166, 185  
 文, 2, 145  
 分割, 183  
 文法エラー, 10  
 プロトタイプオブジェクト, 3, 159  
 プロパティ, 143  
 減らす, 212–214  
 編集画面, v, 2  
 変身する, 31, 201  
 変数, 3, 51, 144  
 ペンあり, 201  
 ペンなし, 36, 200  
 方眼紙, 35, 207  
 本当, 170, 184  
 ボタン, 15, 19, 23, 49, 145, 173, 212  
 ポート, 83  
 マイナーコード, 221  
 マウス, 207  
 混ぜる, 209  
 待つ, 25, 27, 89, 156, 186, 187, 198, 218, 219, 221, 223, 225, 227–229  
 まで, 217  
 右後, 108  
 右回転?, 231  
 右スワイプ?, 231  
 右ダブルクォーテーション, 182  
 右ダブルクォート, 182  
 右二重かぎ括弧, 182  
 右前, 108  
 右回り, 4, 15, 96, 108, 200, 204, 206  
 水色, 16, 171, 208  
 未定義, 122, 160, 176, 192  
 緑, 16, 171, 208  
 無音, 52, 219, 221, 223  
 向き, 200  
 向き?, 203, 205, 207  
 無限ループ, 98  
 紫, 16, 171, 208  
 命令, 2, 8  
 メインスレッド, 155  
 メソッド, 4, 8, 19, 145  
 メッセージ送信, 145  
 メロディ, 48, 218  
 メロディオブジェクト, 48  
 文字消す, 217  
 文字コード, 184  
 文字サイズ, 211  
 文字色, 20, 211  
 文字出す, 217  
 文字列, 146, 165, 169, 170, 175, 182  
 戻る, 200  
 モーター左, 108  
 モーター右, 108  
 指の数?, 230  
 ユーザー定義命令, 103, 107  
 要素数?, 157, 189  
 曜日?, 196  
 横の位置?, 203, 208, 230  
 横向き, 216  
 読む, 113, 157, 189, 192, 193, 198, 212, 213, 215, 229  
 ライントレースカー, 88  
 ラベル, 173, 214  
 乱数, 32, 36, 53, 167, 180  
 乱数初期化, 180  
 ランダムに選ぶ, 189  
 ランダムに作る, 208  
 リスト, 175, 214  
 リターンキー, 115, 120, 174, 213  
 リミットスイッチ, 101, 102, 108  
 リープ, 230  
 リープモーション, 230  
 ルート, 159, 191  
 連結, 183, 190  
 論理積, 184  
 論理否定, 184  
 論理和, 184  
 ローカル版, iii  
 ローカル変数, 148, 163  
 和音, 220  
 割る, 178, 181  
 , 220, 222