5時間でプログラミングを学ぶ: ドリトル編

久野　靖 (筑波大学)

2009.9.18

目 次

第 1 章　プログラミングの第 1 歩 **7**
1.1 コンピュータとプログラム 7
1.2 ドリトルでプログラムを動かす 9
1.3 「だめです」と言われたら… 10
1.4 オブジェクトとメソッド 11
1.5 タートルで絵を描く 11
1.6 さまざまなメソッドとその呼び出し 14
1.7 回数指定の繰り返し 15

第 2 章　複数のものを扱う **17**
2.1 図形オブジェクト 17
2.2 複数の図形を作り出す 19
2.3 図形に色を塗る 21
2.4 規則的に図形を配置する 23
2.5 さいころを振る 25

第 3 章　人とやりとりする **27**
3.1 押しボタンで動作 27
3.2 スライダーで操作 30
3.3 数の計算をする 31
3.4 計算して図形を描く 34

第 4 章　ものを動かして制御する **37**
4.1 枝分かれ 37
4.2 タイマーとアニメーション 40
4.3 衝突時の動作 42
4.4 ゲームに仕立てる 44

第 5 章　プログラムの計画と設計 **47**
5.1 プログラムの計画 47
5.2 条件を持つ繰り返し 48
5.3 配列とその扱い 51
5.4 プログラムの設計 54
5.5 設計に基づくプログラムの作成 55

はじめに

みなさんは、身近なところに**コンピュータ**があるのを、見たことがあると思います。学校のコンピュータ室のパソコン? もちろんそうですが、もっと身近なところにもあります。たとえば、携帯電話機の中や、家庭用ゲーム機の中にもコンピュータが入っています。

では、コンピュータの特徴は何でしょう? それは、「ソフト (**ソフトウェア**) 次第で、何にでも化ける」ということです (図 1)。たとえば、パソコンの上でお絵描きソフトを動かせば、絵が描けます。WWW のソフト (ブラウザ) を動かせば、ネットサーフィンができます。ワープロソフトを動かせば、綺麗な文書が作れます。

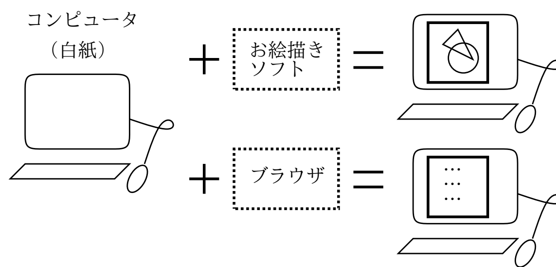

図 1: コンピュータとソフトウェア

ゲーム機でも、ゲームソフト次第で、シューティングゲームも、ロールプレイングゲームも、パズルもできますね。携帯電話でも、自分でソフトを取り替えている感じはしませんけれど、電話している時とメールを打っている時では、別のソフトが動いています。このように、コンピュータそのものは「白い紙」と同じで、そこでどんなソフトを動かすかで、何ができるかが決まってくるのです。

では、ソフトウェアはどうやってできているのでしょう? たとえば、ゲームだったら、そのゲームに登場するキャラクタや背景などの絵が必要ですが、それだけでは足りませんね。キャラクタをさまざまに「動かす」ことが必要です。それも映画やビデオと違って、あなたがゲーム機をさまざまに操作するのに応じて、さまざまに「動かす」必要があります。このような「動きかた」を決めるものを**プログラム**と言います。

つまり、ソフトウェアの中心にはプログラムがあり、それとプログラムが

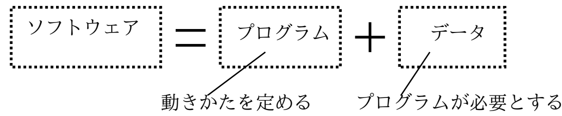

図 2: ソフトウェアのなかみ

動くのに必要なデータなど (ゲームの場合は必要な絵とか) をあわせた全体がソフトウェアなのです (図 2)。

本書は、このソフトウェアをどうやって作るか体験してもらって「プログラムってこういうものかぁ」と思ってもらうための本です。今まで皆さんは、色々なソフトを使ってコンピュータを便利に、快適に、使ってきたではずです。今度は、自分でソフトウェアを作ることに、ちょっと挑戦してみませんか? といっても、自分にそんなすごいことができるだろうか、などと構える必要はありません。(誰かが作った既成のソフトウェアのお世話になる代わりに) まずは白紙のコンピュータを、あなた自身が自分の手で直接動かしてみる、ということなのです。「へえ〜、そんなことができるの?!」って、何だかワクワクしませんか? 最初は大したことはできなくても、きっと既成のソフトウェアを使うのとは違った満足感が得られると思いますよ。それに、プログラムを作ることは、ちょっと頭を使いますが、パズルのようでとても楽しいことです! 本当かどうかは、とりあえず 1 章をやってみれば分かります。では善は急げ、すぐにはじめましょう。

第1章　プログラミングの第1歩

この章では、とにかくプログラムを作りはじめながら、プログラミング言語とはどういうものか、プログラミングするとはどういうことかを感じてみていただきましょう。

1.1　コンピュータとプログラム

コンピュータはとても杓子定規です。それは皆さん、さまざまなところで実感しているでしょう。

たとえば、JR の指定券は窓口でも買えますが、最近では自動販売機 (つまりコンピュータ) でも買えます。窓口であれば、「だいたいこういう切符」と頼んで、もし正確でないところがあれば聞き返してくれますし、満席なら他のおすすめを調べてくれたりもします。

図 1.1: 窓口と自動販売機

一方、自動券売機だと、何月何日の何時のどこ行きの列車、というふうにきっちり情報を指定してやる必要がありますし、寝台列車など扱っていない列車もあります。不便ですが、そのかわり人手を煩わせないので、台数が多くて待たずに済みますし、「今は買わないけれどどの列車はまだ空いているかな」などと調べてみたりもできます。つまり、「杓子定規による不便さ」と「自動化による便利さ」を引き換えにしているわけです。

実は、コンピュータを動かすためのプログラムも、これと似た面があります。つまり、人に何かをたのんでやってもらうのと比べて、コンピュータにプログラムで指示する場合は、次の点が違っています:

- 書き方の規則がきっちり決まっている。

- どのように書いたらどのように動作するかがきっちり決まっている。

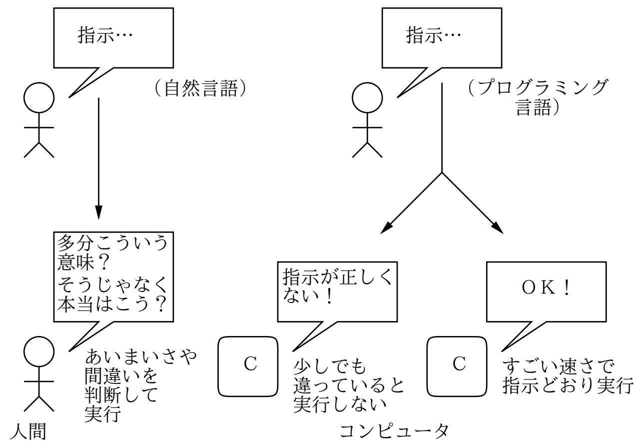

図 1.2: プログラミング言語の特性

つまり、人にものをたのむ時は、多少あいまいな言い方でも意味が通じますし、たのんだことに多少間違いがあっても相手がうまく補ってこなしてくれたりします。でも、コンピュータにたのむ時は「少しでも意味が通らなければ動作しない」「たのんだ内容に間違いがあってもそのまま動作してしまう」という性質があります (図 1.2)。最初にいった杓子定規、というのはここでも成り立っているのです。

これは不便なようですが、逆にそうなっているからこそ…動作するたびにあいまいさや間違いについて考えたりしないようになっているからこそ、コンピュータは、人間には真似できないほど速く、正確に、沢山の動作をこなすことができるのです。

そしてこの、プログラムを書くときの、きっちり決まった書き方のことを、「プログラムするためのことば」という意味で**プログラミング言語**と呼びます。プログラミング言語で書かれた動作命令は、そのプログラミング言語の**処理系**と呼ばれるソフトが、最終的にコンピュータを直接動かす命令へと変換してくれます。だからソフトウェアを作るときは、プログラミング言語によるプログラムを考えて書いていけば良いわけです。

人が普段会話したり手紙でやりとりするためのことば (自然言語) として、日本語、英語など多くのものがあるように、プログラミング言語にもいろいろな種類のものがあります。この本では**ドリトル**という、教育むけに作られたプログラミング言語を用いて、具体的にプログラムを作りながら、コンピュータにさまざまなことをさせてみましょう。

1.2　ドリトルでプログラムを動かす

ドリトルでプログラムを作るには、まずドリトルの処理系を動かします (動かし方が分からない人は付録を見てください)。ドリトルのソフトウェアは、上の方にあるタブで**編集画面**と**実行画面**が切り替えられるようになっています。実際に切り替えてみてください。編集画面では、プログラム (命令のならび) を打ち込んだり、動かしてみて違っていたら手直ししたりします。実行画面では、プログラムが動いた結果の絵などを見ることができます。

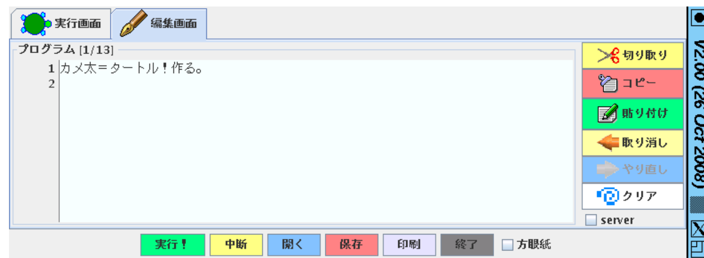

図 1.3: ドリトルの編集画面

ではさっそく、プログラムを打ち込んで動かしてみましょう。編集画面に切り替え、入力欄 (行番号 1 の隣) をクリックして、とりあえず次のような**コード**を打ち込んでください (図 1.3)。コードというのは、プログラムの断片、という意味のことばです。

```
カメ太＝タートル！作る。
```

ちなみに、文字が「=」(半角) か「＝」(全角) かとか、「。」(読点) か「.」(ピリオド) かとか気になる人もいるかも知れませんが、ドリトルではこれらの文字はどちらでも同じに動作するようになっています。

さて、打ち込み終わったら、この状態で「実行！」ボタンを押すと、自動的に編集画面に切り替わってプログラムが実行開始され、画面の中央にカメの絵が現れます (図 1.4)。これでぶじ、最初のプログラムが実行できました。

なお、カメの絵が現れる場所の**座標** (場所を 2 つの数で表したもの) は (0,0) になっています。ここを**原点**と言います。後で場所を座標で指定するときには、ここが基準になるので、おぼえておいてください。

練習 1　ドリトルを動かし、先に説明した 1 行のプログラムを打ち込んで、実行させてみなさい。

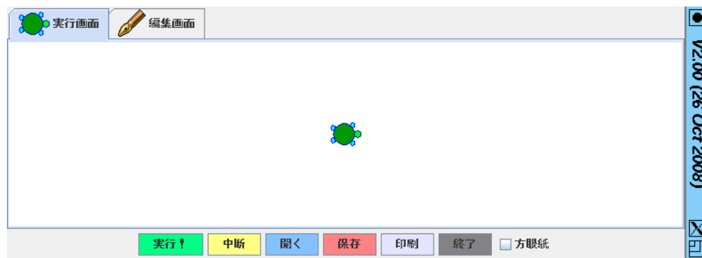

図 1.4: ドリトルの実行画面

1.3 「だめです」と言われたら…

プログラムを動かそうとしても、先に説明したようにスムーズには行かない人もいると思います。前に説明したように、コンピュータはプログラムの書き方に関してはとても杓子定規です。だから、人間なら「これくらいいいじゃない」と思うような違いでも「だめです」と言われてしまうのです。

たとえば、打ち込んだプログラムが次のようになっていたとします。

カメ太 タートル！作る。

すると、実行ボタンを押した時に次のようなメッセージが出ると思います。

```
エラーがあります
1 行目の 5 文字目の「★」の近くまたは前後の行を調べてみてください
カメ太 ★タートル！作る
```

たしかに、「＝」が抜けているので、★の近くにまちがいがあったわけです。では、次のプログラムはどうでしょうか。

カメ太＝タートル！作る

問題なさそうですか? 実行ボタンを押すと次のメッセージが出ます。

```
エラーがあります
1 行目の 12 文字目の「★」の近くで「。」が抜けていないか調べてみてく
ださい。または次の行が正しいか調べてみてください
カメ太＝タートル！作る★
```

言われてみれば、たしかに最後の「。」がありません。ドリトルのプログラムでは「。」は 1 つの命令の終わりを表すので、「。」がないと命令が完結していないことになってしまいます。では、次のは?

カメ太＝タ～トル！作る。

今度は何も悪くなさそうですか? 実行してみると次のように言われます。

```
エラーがあります
1 行目の 6 文字目の「★」の近くまたは前後の行を調べてみてください
カメ太＝タ★〜トル！作る。
```

ここで★のところをよく見ると、音をのばす記号「ー」が「〜」(波記号) になっています。波記号と音をのばす記号は別のものなので、確かにこれでは正しくないのです。

なんて細かくて面倒くさい! と思ったでしょうか。杓子定規というのは、具体的にはこういうことなのです。そして、プログラムを書くということは、このコンピュータの杓子定規さ (よく言えば正確さ) を逆に活用することなのです。そういうわけですから、プログラムを動かそうとして「だめです」と言われたら、「しょうがないなあ」と思いながらコンピュータに分かるように直してやってください。馴れればすぐ直せるようになると思います。

練習 2 先に説明した 1 行のプログラムをわざと間違えたものに直してみて、実行させるとどのように言われるか試してみなさい。

1.4 オブジェクトとメソッド

今日、コンピュータでは非常にさまざまなことができるので、プログラミング言語にとっても、このさまざまなことをうまく書きあらわせることが重要になっています。その 1 つの方向が、さまざまな機能を持った「もの」(**オブジェクト**と呼びます) をうまく扱えるように工夫されたプログラミング言語 (**オブジェクト指向言語**) です。ドリトルもオブジェクト指向言語の 1 つです。

オブジェクト指向言語では、さまざまな「もの」(オブジェクト) に対して、その「もの」が持つ機能 (**メソッド**) を呼び出す形でプログラムの動作を記述します。

たとえば、「赤ペン」に対して「どこまで移動する」「どこまで線を引く」というメソッドを呼び出すことで線を引くことができます。ここで別のオブジェクト「青ペン」に対して同様に呼び出すと、同じように線が引けますが、線の色が違っています。さらに「修正ペン」だと線が引けるのではなく消えるかも知れません (図 1.5)。

このように、「どのオブジェクトの」「どんな操作」という組合せを変えることでさまざまな機能が使い分けられる、というのがオブジェクト指向言語の特徴です。

1.5 タートルで絵を描く

この章ではこれから、ドリトルの**タートルグラフィクス**と呼ばれる機能を使って絵を描いていきます。タートルグラフィクスというのは、画面上を動

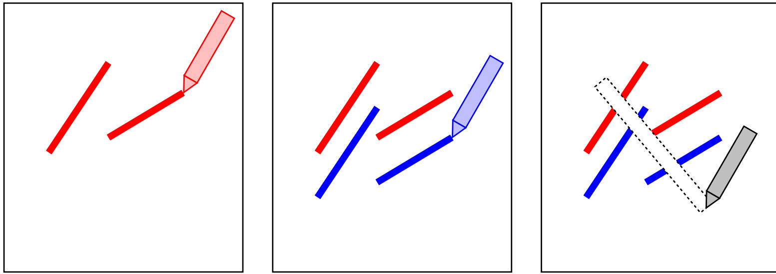

図 1.5: オブジェクトとメソッド

く「もの」(カメなどの動物を思い浮かべてください) にどれだけ動けとか曲がれとか命令し、動いた軌跡で絵が描ける、というものです。そのために、**タートル**オブジェクトを作り出して、それが持つ動作を使っていきます。

具体的に見て行くために、先に動かしたプログラムの内容を説明しましょう。プログラムは次の通りでしたね。

```
カメ太＝タートル！作る。
```

ドリトルの世界では、あらかじめいくつものオブジェクトが用意されていて、それぞれ名前がつけられています。「タートル」というのもそのようなオブジェクトの 1 つです。そして、オブジェクトに何かの動作をさせるには、先に説明したようにメソッドを呼び出します。ドリトルではこれは、次のような形で書きます。

```
オブジェクト！命令
```

ここではタートルオブジェクトの「作る」というメソッドを呼び出すことで、新しいタートルオブジェクトを作っています。そして、この新しいオブジェクトをこれから何回も使うので、次のようにしてそれに名前をつけています。

```
名前＝ … 。
```

最後の「。」は命令の終わりをあらわすために必要なのでしたね。

ではいよいよ、プログラムをもう少し拡張していきましょう。また編集画面に切り替え、最初の 1 行の下に次のように行を書き足して動かしてみてください。実際に試すときは、3 行いっぺんに書き足すのではなく、1 行書き足して、動かしてみて、OK なら次の行を書き足して、動かしてみて、というふうにしてください。

```
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。
```

このプログラムでは、「歩く」や「左回り」という新しい命令が出て来ています。また、そのときに「どれだけ」歩いたり回ったりするかの指定 (**パラメータ**と言います) を、命令の前に書くようになっています。これらはたとえば、「100 歩」歩いたり「90 度」左に回ったりする、というふうに考えればよいでしょう。ちなみに、もちろん「右回り」という命令もあります。

上のプログラムを動かすと、図 1.6 のような絵が描かれます。つまり、タートル「カメ太」が 100 歩歩き、90 度左に回り、さらに 100 歩歩いた時の軌跡ができているわけです。なお、「1 歩」というのは画面を構成する小さな点 (ピクセル) の大きさが単位ですが、とにかく動かしてみてこの長さで 100 歩ぶんなのだと思ってください。

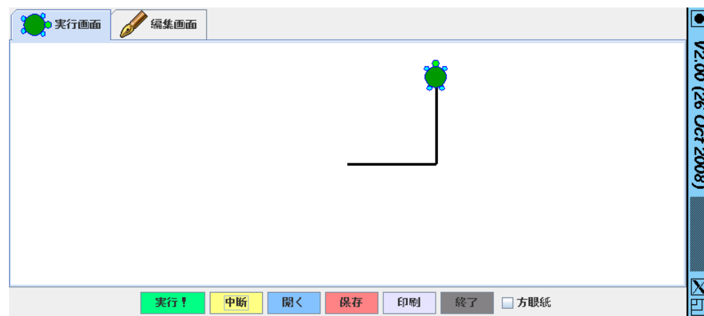

図 1.6: 途中で折れ曲がった線を引く

ところで、「なぜタートルに直接命令しないで、カメ太というものを作っているのか」と思われるかも知れません。これは、直接使ってしまうとタートルがその 1 個になってしまうからです。今は 1 個でいいかも知れませんが、これから色々なプログラムを作っていくと、タートルが複数あった方が便利なことがあります。そのためドリトルでは、オブジェクトを使う時にはまずそのオブジェクトを作って名前をつけ、それからそれを使う、というふうになっているのです。

演習 3 ドリトルのプログラムで、図 1.7(a)〜(c) と (だいたい) 同じ図形を描いてみなさい。

演習 4 紙に 5 本くらいまでのつながった線だけでできた (ひとふで描きの) 簡単な図形を描いてみなさい。それと (だいたい) 同じ絵を、ドリトルのプログラムで描いてみなさい。友人と課題を出しあってみてもよいです。

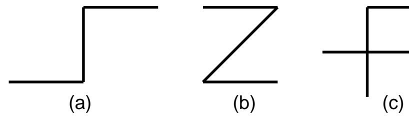

図 1.7: 練習問題の図形その１

1.6　さまざまなメソッドとその呼び出し

タートルオブジェクト (と、先の例の「カメ太」のようにタートルから作りだしたもの) は、画面に絵を描いたりするためのいくつかの命令 (メソッド) をあらかじめ持っています。主要なメソッドとその機能を表 1.1 に挙げておきます。

表 1.1: タートルのメソッド

名前	機能	サンプル
作る	新しいタートルを作る	カメ太＝タートル！作る。
歩く	前に進む	カメ太！１００　歩く。
戻る	後ろに戻る	カメ太！１００　戻る。
右回り	右に指定角度だけ回る	カメ太！９０　右回り。
左回り	左に指定角度だけ回る	カメ太！９０　左回り。
移動する	右に x、上に y 移動	カメ太！１０　２０　移動する。
位置	指定した位置 (x, y) に移動	カメ太！５０　－５０　位置。
ペンあり	線を引くようになる	カメ太！ペンあり。
ペンなし	線を引かないようになる	カメ太！ペンなし。
中心に戻る	画面の中心に移動	カメ太！中心に戻る。
閉じる	描き始めの点に移動	カメ太！５０　歩く　３０　右回り　５０　歩く　閉じる。
線の色	線の色を変更する	カメ太！（緑）線の色。
線の太さ	線の太さを変更する	カメ太！５　線の太さ。

このように、ドリトルではどのオブジェクトも、それぞれ固有のメソッドを持っています。そして、タートルから作ったオブジェクトの場合はメソッドは表 1.1 のものでしたが、ほかの種類のオブジェクトはまったく別のメソッドを持っています。つまり、どのようなメソッド (命令) が使えるかは、オブジェクトごとに違っているということです。

ところで、ドリトルでは多くのメソッドは値 (オブジェクト) を返します。そして、その返したオブジェクトにさらにメッセージを送る場合は、いちいち「！」を書かなくても次のように続けて書くことができます。

```
オブジェクト！… 命令₁ … 命令₂ … 命令₃。
```

表 1.1 のメソッドの場合、最初の「作る」以外は、もともとのメソッドの受け手 (先のプログラムの例でいえば「カメ太」がそのまま返されます。ですから、先の例題プログラムは次のように 2 行で書くこともできます。

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

このようにプログラムは、まったく同じことをするものでも、何通りものやり方で書くことができます。では、どのように書くのがよいのでしょう? それは、なるべく明快で分かりやすく書くのがよいのです。というのは、プログラムはもともと、ごちゃごちゃで難しくなりやすいので、少しでも分かりやすくしておかないと扱えなくなってしまうからです。

演習 5 前の問題で作ったプログラムを短く書き直してみなさい。自分ではどのようにするのが見やすいと思うか考えてみるとよいでしょう。

演習 6 図 1.8 と同じような図形を描いてみなさい。(ヒント: (a) – 「ペンあり」「ペンなし」を使う、(b) – 「図形を作る」で切り離した後で「線の太さ」を使う、(c) – 「閉じる」を使う、(d) – 「中心に戻る」を使う)

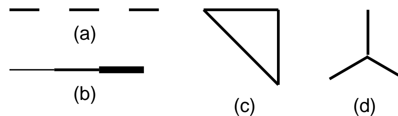

図 1.8: 練習問題の図形その 2

演習 7 前の問題と同じように、ただし今度は新しく出て来たメソッドも使って、線だけでできた簡単な図形をドリトルで描いてみなさい。つながっていない図形にしたり、線の太さや色を変えてみたりしてみてもよいでしょう。

1.7 回数指定の繰り返し

繰り返しとは文字通り、プログラムの一部分を繰り返し実行することを言います。ここでは、一番簡単な繰り返しとして、指定した回数だけ繰り返すやり方を見てみましょう。

```
カメ太＝タートル！作る。
「カメ太！１００　歩く ３０　左回り」！３ 繰り返す。
```

このように、動作の繰り返しなどを指定するときは、まず、その動作を「…」で囲みます。上の例では1行だけですが、何行あっても全体を『「』と『」』で囲めばよいのです。この、一連の動作を囲んだものを、ドリトルでは**ブロック**と呼んでいます。そして、ブロックもオブジェクトの1種で、さまざまなメソッドを持っています。そのなかに、「繰り返す」というメソッドがあって、上の例では回数として「3」を指定してこのメソッドを呼び出しているわけです。

このプログラムを動かしたようすを図1.9に示します。見てわかるように、100歩あるいて、30度曲がって、100歩あるいて、30度曲がって、さらに100歩あるいて、30度曲がっています (カメ太は最終的に90度曲がったので、真上を向いていることに注意)。このように、繰り返しを使うと、短いプログラムでも沢山の動作が指定できます。

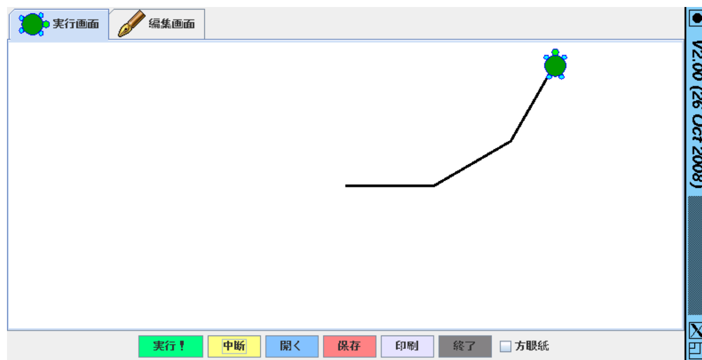

図 1.9: 途中で折れ曲がった線を引く

演習 8 例題プログラムを手直しして、1辺の長さや、曲がる角度や、繰り返す回数をさまざまに変えて試してみなさい。

演習 9 繰り返しを使って、図1.10のような図形を描いてみなさい。

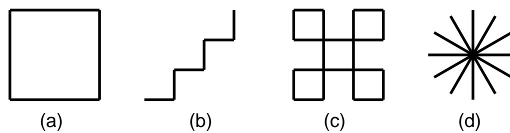

図 1.10: 練習問題の図形その3

演習 10 規則性のある絵を紙に描き、それをドリトルの繰り返しを使って描いてみなさい。友達と課題を出しあってもよいです。

第2章　複数のものを扱う

この章では、前の章で描いた図形をもとにオブジェクトを作り、複数のオブジェクトにそれぞれ名前をつけて扱い、色を塗ったり、組み合わせるやり方を体験していただきましょう。

2.1　図形オブジェクト

前の章では、タートルオブジェクトを作り、「歩く」「右回り」などのメソッドを呼び出すことで、画面にさまざまな形を描きました。たとえば、次のプログラムでは、正方形ができますね。

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

この正方形は、タートルオブジェクトであるカメ太の「歩いたあと」として画面に見えているのですが、タートルオブジェクトのメソッド「図形を作る」を使うことで、この「歩いたあと」を切り離して１つの**図形オブジェクト**を作ることができます。プログラムを見てみましょう。

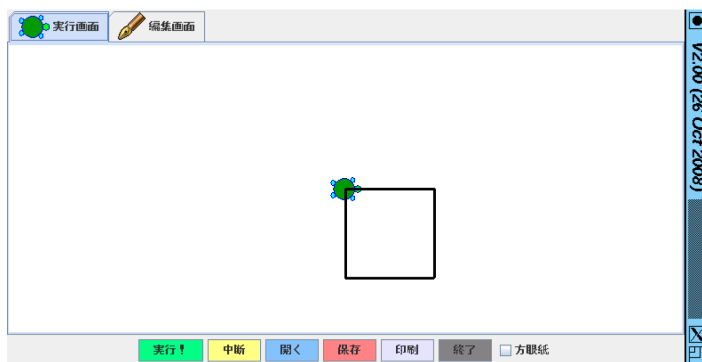

図 2.1: 正方形を図形にしたところ

```
カメ太＝タートル！作る。
「カメ太！１００　歩く　９０　右回り」！４　繰り返す。
箱＝カメ太！図形を作る。
```

このように、四角を描いた後に、カメ太に対して「図形を作る」というメソッドを呼び出すと、歩いたあとが図形となって返されます。図 2.1 がこれを実行したようすですが、よく見ると歩いた後がカメから切り離されて見えている (図形の線がカメの上にある) のがわかります。なお、「箱＝」というのは、カメ太を作った時と同じように、作った図形オブジェクトに名前をつけ、後からそれと指定できるようにするためのものです。このような名前の意味については、少し後で説明します。

四角が図形オブジェクトになったら (そして「箱」という名前がついたら)、どんな利点があるのでしょうか? それは、今度はその「箱」に対して、「移動する」「左回り」などのメソッドにより、その図形に対してさまざまな指示が与えられる、ということです。例を見てみましょう。

```
カメ太＝タートル！作る。
「カメ太！１００　歩く　９０　右回り」！４　繰り返す。
箱＝カメ太！図形を作る。
箱！１００　５０　移動する。
箱！３０　左回り。
```

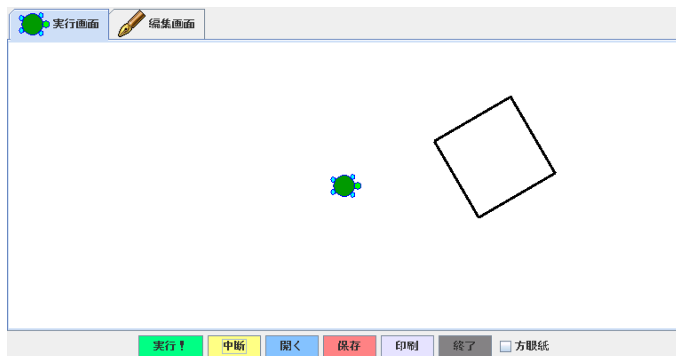

図 2.2: 図形を動かして回転させたところ

これを動かしたところを図 2.2 に示します。確かに、正方形が右に 100 歩ぶん、上に 50 歩ぶん移動し、30 度左に回転しています。

ところで、図形オブジェクトのメソッド「移動する」や「右回り」は、元の図形オブジェクトをそのまま返すので、上のプログラムの最後の 2 行は次のように 1 行にしても構いません。見やすい方を使ってください。

```
箱！１００　５０　移動する　３０　左回り。
```

図形オブジェクトのメソッドを表 2.1 に示します。「左回り」「右回り」「移動する」「位置」などはタートルの同じ名前のメソッドと同様に働きますが、タートルのメソッドではタートルを操作するのに対し、図形オブジェクトの

メソッドでは図形オブジェクトを操作することに注意してください。なお、色を塗る方法については少し後で説明します。

表 2.1: 図形のメソッド

名前	機能	サンプル
図形を作る	タートルの歩いたあとから図形を作る	カメ太！図形にする。
右回り	右に指定角度だけ回る	箱！３０　右回り。
左回り	左に指定角度だけ回る	箱！３０　左回り。
移動する	右に x、上に y 移動	箱！１０　２０　移動する。
位置	指定した位置 (x, y) に移動	箱！５０　－５０　位置。
塗る	色を指定してその色で塗る	箱！（緑）塗る。
拡大する	指定した倍率で拡大/縮小	箱！０．５　拡大する。

演習 1 上の例題プログラムを動かし、正方形の位置や角度をさまざまに変えてみなさい。また、拡大/縮小させてみなさい。図形は正方形ではない別の形でもかまいません。

2.2 複数の図形を作り出す

図形が 1 つだけでは、あまり面白い絵はできないので、こんどは図形を複数にしてみましょう。図形を複数にするのには、次の 2 とおりの方法があります。

1. タートル (カメ太) に図形を描かせ「図形を作る」で作ることを複数回おこなう。
2. 既に作った図形に対し、メソッド「作る」を呼び出すことで、図形をコピーする。

実際にやってみましょう。次のプログラムは、三角形と正方形を組み合わせて屋根のついた家の形を作り、さらに正方形を増やしています。

```
カメ太＝タートル！作る。
「カメ太！１００　歩く　１２０　左回り」！３　繰り返す。
三角＝カメ太！図形を作る。
「カメ太！１００　歩く　９０　右回り」！４　繰り返す。
箱１＝カメ太！図形を作る。
箱２＝箱１！作る　１２０　０　移動する。
箱３＝箱２！作る　１２０　０　移動する　１５　左回り。
```

「箱２」は家の本体になる「箱１」をコピーした後、右に 120 だけ動かしています。「箱３」は「箱２」をさらにコピーした後、右に 120 動かし、15 度回転させています (図 2.3)。

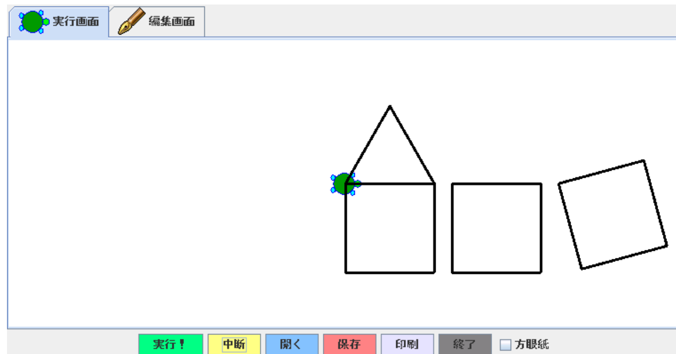

図 2.3: 三角形と四角形を組み合わせる

ところで、それぞれの正方形に「箱１」「箱２」のように別の名前をつけているのはなぜでしょうか。実はこのプログラムの場合でいえば、次のように全部「箱」のままでも構いません。

```
…略…
箱＝カメ太！図形を作る。
箱＝箱！作る　１２０　０　移動する。
箱＝箱！作る　１２０　０　移動する　１５　左回り。
```

ただし、2 個目の「箱＝」のところで、新しく作った正方形に「箱」という名前を使ってしまうので、最初の正方形を指す名前はなくなります。さらに、3 個目の「箱＝」で、最後に作った正方形に「箱」という名前を使うので、2 番目の正方形を指す名前もなくなります。ですから、この後さらに 1 番目や 2 番目の正方形を動かそうとしても指定できなくなるわけです。

ここまで出て来た「カメ太」「三角」「箱」のようなオブジェクトをさす名前のことを、ドリトルでは**変数**と呼んでいます。変数に対して「＝」を使うと、変数は新しいオブジェクトを指すように変化します。つまり、「＝」は変数に別の値を入れることを意味しているのです。

上の「箱」の例のように、1 つの変数に次々に新しいオブジェクトを入れていっても構いませんが、新しいものを入れると前に入っていたものは忘れられてしまいますから、変数に入れたものを後で使いたい場合は、別のものを入れないように注意する必要があります。そのためには、「箱１」「箱２」のように名前を分けるのがいいわけです。

演習 2 複数の図形を含んだ絵を描いてみなさい。いきなりプログラムを作るのでなく、どのような絵にするか、紙に描いて計画し、それに従って作るようにすること (完全に計画通りでなくても構いません)。

2.3 図形に色を塗る

表 2.1 には色を塗るメソッド「塗る」が含まれていますが、色の指定方法をまだ説明していませんでした。ここで説明しましょう。ドリトルではすべての値はオブジェクトなので、色も**色オブジェクト**になっています。色オブジェクトを使う時は、次のどれかを利用します。

1. 予め色を入れてある変数名「黒、赤、緑、青、黄色、紫、水色、白」のどれかを指定する。

2. 「色」オブジェクトのメソッド「作る」を使い、光の三原色である赤、青、緑の成分の強さを指定して色を作り出す。

3. 既にある色に対してメソッド「明るくする」「暗くする」「半透明にする」を呼び出して新しい色を作る。

4. 「光」オブジェクトや「絵具」オブジェクトのメソッド「混ぜる」を複数の色を指定して呼び出すことで、複数の色が混ざった新しい色を作る。

1 については、使いたい色の変数名を指定するだけです。

2 の色の作り方ついては、もう少し説明しましょう。三原色をさまざまに混ぜ合わせることでどのような色でも作れることは知っていますね? 「色！ r g b 作る」というメソッド呼び出しは、光の三原色である赤、青、緑の強さを 0〜255 の数値で r、g、b のところに指定することで、さまざまな色を作り出させてくれるのです。3 と 4 は、例題の中で説明しましょう。

```
カメ太＝タートル！作る。
「カメ太！１００　歩く　９０　左回り」！４　繰り返す。
箱１＝カメ太！図形を作る（青）塗る　－３００　５０　位置。
箱２＝箱１！作る　１２０　０　移動する。
色３＝色！２５０　２００　２００　作る。
箱３＝箱２！作る（色３）塗る　１２０　０　移動する。
箱４＝箱３！作る（色３！暗くする）塗る　１２０　０　移動する。
箱５＝箱４！作る（緑！半透明にする）塗る　４０　－２０　移動する。
色６＝光！（赤）（青）混ぜる。
箱６＝箱４！作る（色６）塗る　０　－１５０　移動する。
色７＝絵具！（赤）（青）混ぜる。
箱７＝箱６！作る（色７）塗る　１２０　０　移動する。
```

この例題では、図形は正方形だけで、それを次々にコピーしてさまざまな色を塗っています。最初にカメ太で正方形を描き、それを図形にして青で塗り、左上の位置に移動します。これを箱 1 に入れていますい。

ところで、色を塗るところで「（青）塗る」のように「青」がかっこで囲んでありますが、これはなぜでしょうか? それは、ドリトルでは「！」の右側にある名前はメソッド名として扱うので、変数の値を取り出して使いたいときはメソッド名ではないことを示すために（…）で囲む必要があるのです。な

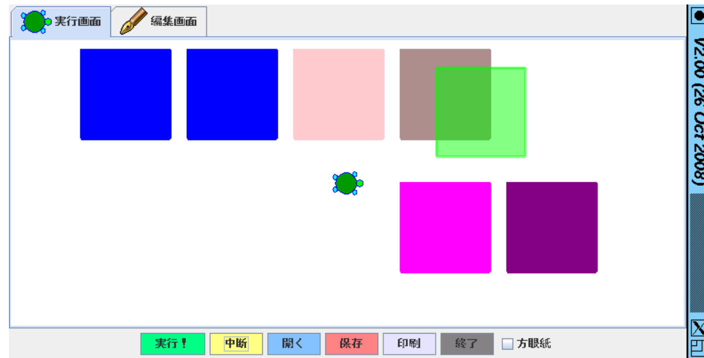

図 2.4: さまざまな色の例題

お、(…) の中には変数名だけでなく、さまざまな計算式やメソッド呼び出しも書くことができます。

まてよ、これまでの「１００　歩く」の「１００」はどうなのでしょう? それは、名前ではなく数字ですから、命令と間違う心配はないので、かっこで囲まないでそのまま書けるのでした。お望みなら、「(１００) 歩く」のようにかっこで囲んでも構いません。

例題の続きですが、箱 2 は箱 1 をコピーして移動しただけです。色などもそのままコピーされるので、箱 2 も青色になります。

次は、新しい色を作ってみています。色 3 は、三原色の強さを指定して作る方法で指定しました。具体的には、赤が最大に近く (最大は 255、最小は 0 です)、緑と青はやや明るめに光った色で、色としては肌色になります。箱 3 は箱 2 をコピーしてこの色に塗ったものです。

箱 4 は、色 3 を暗くした色で塗ってあります。箱 5 は、緑を半透明にした色で塗ってあります。このように、(…) の中に新しい色を作り出す命令を書いてもいいのです。半透明の場合はもちろん、重ねると下の絵が透けて見えます。

箱 5 と箱 6 の色は、どちらも赤と青を混ぜたものですが、箱 5 では光として混ぜているので、マゼンタになります (光として混ぜると、明るさが増します)。箱 6 では絵具として混ぜるので、ご存じのとおり、紫になります (絵具として混ぜると、元の色より暗くなります)。

演習 3 演習 2 で描いた図形に、好きな色を塗ってみなさい。

演習 4 正方形を 6〜8 個並べて、それぞれに色を塗りなさい。全体として、単調すぎず、バラバラ過ぎないように、図形の配置と色を工夫してみなさい。

2.4　規則的に図形を配置する

先の例題のように、図形が多くなると、それを作るための命令や図形を保持しておくための変数も多くなり、プログラムが複雑になりがちです。でも、それを避ける方法もあります。それは…みなさんも既に知っている方法です。四角を書く時に、「１００　歩く　９０　右回り」を 4 回書くかわりに「繰り返す」を使いましたね。図形を増やすのにも、同じように「繰り返し」が使えるはずです。やってみましょう。

```
カメ太＝タートル！作る。
「カメ太！１００　歩く　９０　左回り」！４　繰り返す。
箱＝カメ太！図形を作る　－３００　０　位置。
「箱＝箱！作る。
  箱！１００　０　移動する　１０　右回り」！４　繰り返す。
```

このプログラムでは、1 つ正方形を作って、画面の左下に置いた後、「繰り返し」を使ってそれをコピーし、右に動かし、回転させることを 4 回行います。これによって、合計 5 つの正方形が、違う場所に違う角度で現れるわけです (図 2.5)。

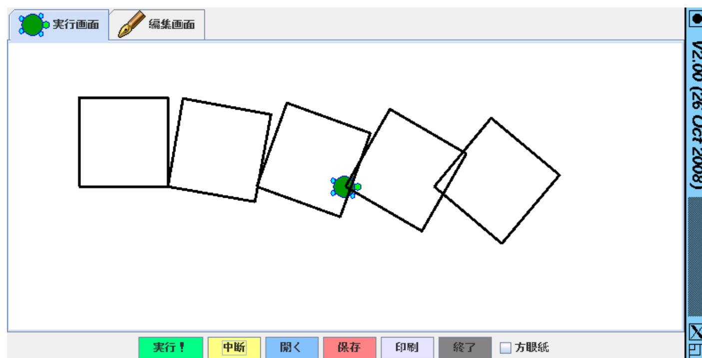

図 2.5: 繰り返しで図形を配置

ところでこのプログラムは、前に避けていたこと、つまり 1 つの変数「箱」を複数の正方形のために使っている、という性質があります。それが一番はっきりしているのは、次のところです。

```
「箱＝箱！作る。
  箱！１００　０　移動する　１０　右回り」！４　繰り返す。
```

この場合、「箱！作る」のところの「箱」はコピーする前の正方形です。そして、コピーしてできた正方形を「箱＝」で変数「箱」に入れますから、それ以後はコピーする前の正方形を指定することができなくなります。2 行目の「箱」はコピーしてできた新しい正方形を意味していて、その位置や角度を変えているわけです。

そして、これらの部分が 4 回繰り返されますから、次の繰り返しで「箱＝箱！作る」のところを実行すると、また新しい正方形が作られ、さっきの正方形は指定できなくなり、それが次々と続いていくわけです。

なぜ、前は避けていた、同じ名前の使いまわしをやっているのでしょうか。それは、ここでは決まった短いコード (ブロック) を何回も繰り返すことで多数の図形を作り出そうとしているのですから、その短いコードの中にある変数には繰り返しごとに新しいものを入れていくしかない (そうしないと多数のものが作れない) わけです。

ところで、「繰り返す」の対象になるブロックの中で「今は何回目の繰り返しか」が分かると便利なことがあります。それには、ブロックを次のような形にします。

```
「｜ｉ｜　…　」！４　繰り返す。
```

この、冒頭部分の「｜…｜」で囲んだ名前のことを**パラメタ**と呼びます。「繰り返す」でパラメタをつけると、ブロックの中では、その値は 0、1、2、3 と変化します (たとえば繰り返しの数が 100 なら、0〜99 まで変化します。0 から数え始めるのは、コンピュータの流儀ではこういう風習だからです)。

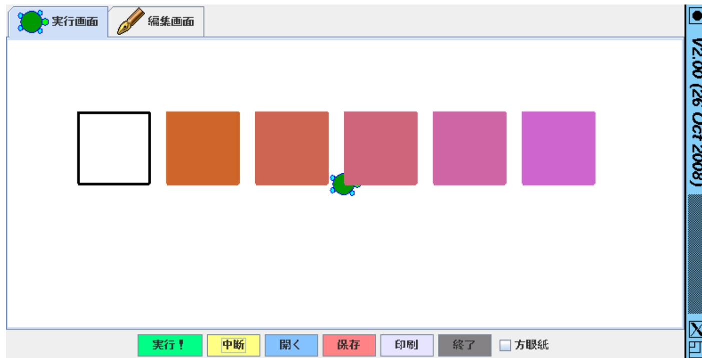

図 2.6: 繰り返しで色を変化

では、このパラメタｉを元にして色合いのうち青色の強さを変化させてさまざまな色の正方形を作ってみましょう (図 2.6)。

```
カメ太＝タートル！作る。
「カメ太！８０　歩く　９０　左回り」！４　繰り返す。
箱＝カメ太！図形を作る　－３００　０　位置。
「｜ｉ｜
　箱＝箱！作る。新色＝色！２００　１００　（ｉ＊４０）作る。
　箱！１００　０　移動する（新色）塗る」！５　繰り返す。
```

「＊」は掛け算を表しています。このほかに、「＋」(足し算)、「－」(引き算)、「／」(割り算)、「％」(剰った余り) も使えます。このような計算式を書

く場合もやはり、（…）で囲んだ中に書いてください。この場合は、ｉは繰り返しによって 0、1、2、3、4 と変化しますから、塗る色のうち青の強さは 0、40、80、120、160、200 と変化します (最大が 255 を超えてはいけないことに注意してください。もし超えると、正しい色ができないので、塗っても色が変化しません)。

演習 5 好きな図形を「繰り返し」を使って規則的に配置してみなさい。位置や大きさや角度を好きに変えてみるとよいでしょう。色も規則的に塗ってみるとよいでしょう。

演習 6 「繰り返し」を使って図形を規則的に配置し、図 2.7 のような形を作ってみなさい。

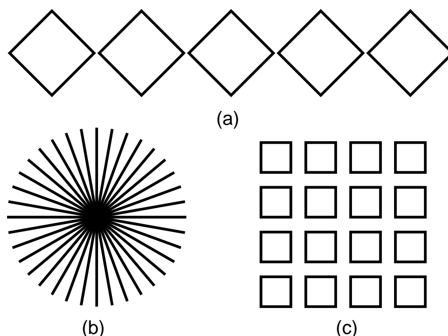

図 2.7: 繰り返しの練習問題

2.5 さいころを振る

規則的なのもいいですが、あまり規則づくめなのも息が詰まります。そこで、さいころの話をしましょう。普通のさいころはころがすと 1〜6 のどれかが出ます。1〜6 というのは、サイコロが立方体 (正 6 面体) だからですが、コンピュータ上でやるのなら 6 に限らなくてもよいはずです。ドリトルでは、「`random(`N`)`」という式で、$1 \sim N$ の範囲の数をランダムに出してくるサイコロが作れます。ちなみに、プログラミングの用語ではサイコロのような「ランダムな数」のことを**乱数**と呼びます。

これを利用して、沢山の正方形を作り、さいころを振ってそれらの位置や色を設定してみることにしましょう。

```
カメ太＝タートル！作る。
「カメ太！８０　歩く　９０　左回り」！４　繰り返す。
箱＝カメ太！図形を作る。
「箱ｘ＝箱！作る（random(600)-300）（random(400)-200）位置。
  新色＝色！(random(255)) (random(255)) (random(255)) 作る。
  箱ｘ！（新色）塗る」！３０　繰り返す。
```

位置は横方向が -300〜300(だいたい。正しくは $-299 \sim 300$ になります) になるように、サイコロで 600 までの数を作り、それから 300 を引いています。縦方向も同様です。色は、1〜255 のサイコロを振って、赤、緑、青の強さを決めています。このプログラムを動かしたところを、図 2.8 に示します。位置や色をサイコロで決めているので、実行させるたびに絵は違ったものになります。

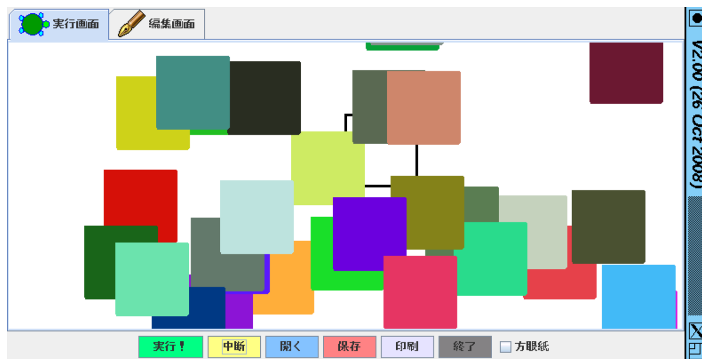

図 2.8: 乱数の例題

演習 7 多角形や線などを組み合わせた「絵」を紙の上に描き、図形を組み合わせてその絵を作るプログラムを作成しなさい。色は自分で計画して指定してもよいですし、乱数で決めてもよいです。

第3章　人とやりとりする

これまでに作ったプログラムでは、実行とともに一気に絵が描かれ、あとは完成した絵を眺めるだけでした。この章では普通のソフトのように、使っている人がボタンなどで操作するようなプログラムを作りましょう。

3.1　押しボタンで動作

みなさんが普段コンピュータを使う時は、画面に表示されているボタンやメニューやスライダーなどをマウスで操作してさまざまな動作を起こしますね? このような方式を「**GUI**(Grapihcal User Interface)」と言います。そして、使う人が操作するボタンやメニューやスライダーなどのことを「**GUI 部品**」と呼びます。

私たちが作るプログラムでも、GUI 部品を用意することで、プログラムを「使う人」がさまざまに操作できるようなプログラムを作ることができます。これから実際に作ってみましょう。

基本的だけれど、とても役に立つ GUI 部品に、**ボタン** (押しボタン) があります。ボタンは、マウスでクリックして「押す」ことができ、押すと何らかの動作を起こすことができるような、GUI 部品です。

図 3.1: プログラムで『前進』ボタンを作ったところ

さっそく、ボタンを作るプログラムを見てみましょう (図 3.1)。ボタンもタートルと同じように「ボタン」に対して「作る」メソッドを呼び出します

が、その時パラメタとして「ボタンに表示させる文字列 (ラベル)」を指定します。**文字列** (文字のならび) は、ドリトルでは文字を『…』または”…”で囲んで表します。次のプログラムでは、『前進』というラベルのついたボタンを作り、それを「前進ボタン」という変数に入れています。

```
前進ボタン＝ボタン！『前進』作る。
```

プログラムを動かし、できたボタンをマウスでクリックしてみてください。確かにボタンがへこんだりして、押せている感じがしますが、何も動作は起きません。それは当然で、自分でプログラムを作っているのですから、「どのような動作が起きる」ということも自分でちゃんと指定しなければ何も起きないのです。

ボタンが押されたときに起こって欲しいことは、ボタンに「動作」という名前のメソッドを定義することで指定します。つまりボタンは、押すことでそのボタンの「動作」というメソッドが動くようにできているのです。

ドリトルでオブジェクトにメソッドを定義するのは、次の形によります。

```
オブジェクト：メソッド名＝「　…起こって欲しいこと…　」。
```

右側の「…」は、すでによく知っているブロックです。これまではブロックは繰り返しのために使ってきましたが、ブロックにはもっと色々な用途があるのでした。

では実際に、先の前進ボタンに「カメ太を前進させる」という動作をつけてみましょう (そのために前進ボタンという名前にしてあったわけです)。もちろん、カメ太もあらかじめ用意する必要があります。

```
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」。
```

なお、メソッドを定義したのですから、「前進ボタン！動作」というコードでそれを実行させることも、もちろんできます。しかし、上のプログラムでは人が押しボタンを押した時に「動作」を実行させるだけなので、「前進ボタン！動作」というコードは含まれていません。

さあこれで、前進ボタンを押すたびにカメ太が前進して線を引くようになりました。前進だけではまっすぐな絵しかできないので、曲がるのもできるようにしてみましょう (図 3.2)

```
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」。
右回りボタン＝ボタン！『右回り』作る。
右回りボタン：動作＝「カメ太！１０　右回り」。
```

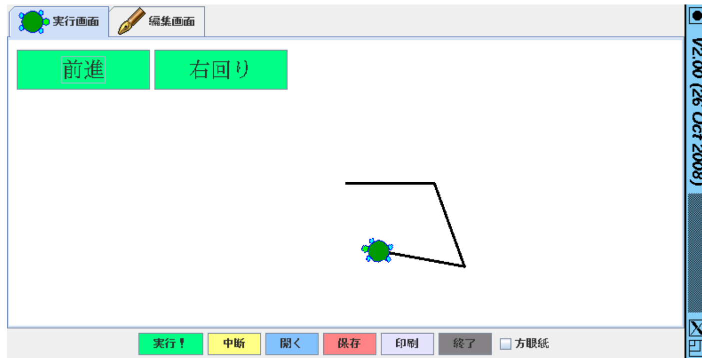

図 3.2: 2 つのボタンで絵を描く

演習 1 上の例題プログラムを動かしてみなさい。うまく行ったら、左回りさせるボタンや、(形を) 閉じるボタンや、そのほか自分で思い付いた機能を持つボタンを作ってみなさい。

演習 2 ボタンなどの GUI 部品オブジェクトも、タートルや図形と同様、位置や色などを設定するさまざまなメソッドを持っています。表 3.1 に、ボタンを含むすべての GUI 部品オブジェクトで共通に使えるメソッドを示しました。これらを活用して、演習 1 のプログラムのボタンの位置や見え方を調整してみなさい。

なお、位置の指定で「－ ５０」などマイナスを指定した場合は、原点 (中心) より左側・上側に５０のところ、という意味になります。

表 3.1: GUI 部品共通のメソッド

名前	機能	サンプル
位置	指定した位置 (x, y) に移動	Ａボタン！５０　－ ５０　位置。
移動する	右に x、上に y 移動	Ａボタン！１０　２０　移動する。
文字サイズ	文字サイズ（ポイント数）を指定	Ａボタン！１６　文字サイズ。
塗る	色を指定してその色で塗る	Ａボタン！(黄色）塗る。
文字色	文字の色を指定する	Ａボタン！(赤）文字色。
大きさ	幅と高さを指定する	Ａボタン！８０　４０　大きさ。

3.2　スライダーで操作

別の GUI 部品として、**スライダー**を使って見ましょう。これは、スライドレバーをマウスでドラグしてさまざまな値を設定できるようなものです。スライダーが持つ値の範囲は 0〜100 で、とくに指定せずに作った場合は最初は値が 50(中央の値) になっています。

今度の例題では、スライダーを使って図形の回転を制御してみましょう。まず「前進ボタン」「右回りボタン」で図形を描き、「作成ボタン」でそれを図形にして、変数「マイ図形」に入れます。入れるところで「：マイ図形＝」のように「：」がついていますが、これは、メソッドの中で単に「マイ図形＝」で変数に値を入れると、そのオブジェクト (例題の場合は「作成ボタン」) の中でだけ使える変数に入ることになっているからです (なぜそうなのかは、また後で説明します)。「：」をつけると、これまで通りどこでも使える変数になります。

```
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」。
右回りボタン＝ボタン！『右回り』作る。
右回りボタン：動作＝「カメ太！１０　右回り」。
作成ボタン＝ボタン！『図形作成』作る。
作成ボタン：動作＝「：マイ図形＝カメ太！図形を作る」。
回転制御＝スライダー！作る　０　値。
回転制御：動作＝「マイ図形！(３．６＊（回転制御！値？))向き」。
```

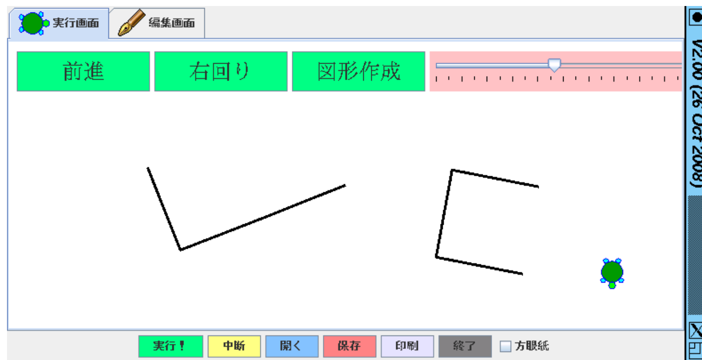

図 3.3: スライダーで回転を制御する

スライダーオブジェクトは他のオブジェクトと同じように「スライダー！作る」で作成し、最初は「回転していない」0 度にしておきたいので、値を 0 に設定します。スライダーもボタンと同様、メソッド「動作」を定義することでスライダーが動いたときの動作が指定できます。ここでは、スライダーが動くごとに、現在のスライダーの値 (0〜100) をメソッド「値？」で読み出

し、0〜360 度の範囲にするために 3.6 倍して、マイ図形の向きとして設定します。このプログラムを動かすと、スライダーが現れ、図形を作った後でスライダーを動かすと、その動きにつれて図形の角度が変わります (図 3.3)。

演習 3 さらに、マイ図形に対して次の操作を行うような GUI 部品を追加してみなさい。スライダーは GUI 部品共通の表 3.1 に示したメソッドのほか、表 3.2 に示した固有のメソッドも持っています。

a. マイ図形の縦方向、横方向の位置を変更するスライダー。

b. マイ図形に黄色など特定の色を塗るボタン。

c. マイ図形に任意の色を塗るボタン。色は赤・緑・青の強さをあらかじめ 3 つのスライダーで設定しておく。

d. マイ図形の大きさを拡大・縮小するスライダー。(注意!「現在何倍になっているか」を変数でおぼえておかないとうまく行かないので、ちょっと面倒です。)

表 3.2: スライダー固有のメソッド

名前	機能	サンプル
値	値を設定する	回転制御!１００　値。
値?	現在の値を取り出す	角度＝３．６＊（回転制御!値?）。
縦向き	スライダーを縦向きにする	回転制御!縦向き。
横向き	スライダーを横向きにする	回転制御!横向き。
文字出す	値の数値を表示する	回転制御!文字出す。
文字消す	値の数値を表示しなくする	回転制御!文字消す。

3.3 数の計算をする

ここまででは、ずっと絵を描いたり操作してきましたが、ここで少しはコンピュータ (英語で「計算する装置」という意味) らしいプログラムも作ってみましょう。

アメリカでは温度を華氏で表しますから、アメリカのテレビ番組などを見ていて温度の数値が出てきても、普段摂氏を使っている私たちには、どれくらい熱いのか寒いのか分かりません。そこで、華氏の温度を摂氏に換算するプログラムを作ってみましょう。

華氏の温度 F を摂氏の温度 C に換算するのには、次の式を使います。

$$C = 5 \times (F - 32) \div 9$$

これを電卓で計算してもいいのですが、引き算、割り算、掛け算が出て来てけっこう面倒です。そこで、画面の入力欄に華氏の温度を打ち込み、「計算」ボタンを押すと摂氏の温度が計算されて表示される、というふうにしてみましょう。

そのためには、使う人が数値を打ち込んだり結果を表示させたりするための GUI 部品が必要です。また、数値だけでは不親切ですから、どこが何を意味しているのかを説明する必要もあります。そのために、ここでは次の2つの GUI 部品を新たに使います。

- フィールド — 数値やその他の文字を、使う人が打ち込んだり、プログラムが書き込んだりできる。また、書き込まれている内容をプログラムが読み出すこともできる。

- ラベル — 説明のために文字を表示しておくことができる。

これらはいずれも、おなじみ「作る」で作り出せ、位置などの設定はボタンなどと同じように表3.1のメソッドで行えます。計算そのものは、これまでと同じようにボタンの動作として定義します。

では、摂氏華氏変換プログラムを見てみましょう (図3.4)。

```
華氏ラベル＝ラベル！『華氏の温度』作る　－２５０　９０　位置。
華氏入力欄＝フィールド！作る　－１００　１００　位置。
計算ボタン＝ボタン！『温度計算』作る　７０　１００　位置。
摂氏ラベル＝ラベル！『摂氏の温度』作る　－２５０　４０　位置。
摂氏出力欄＝フィールド！作る　－１００　５０　位置。
計算ボタン：動作＝「
　華氏＝（華氏入力欄！読む）。
　摂氏＝５＊（華氏－３２）／９。
　摂氏出力欄！（摂氏）書く」。
```

図 3.4: 温度の変換計算プログラム

華氏入力欄と摂氏出力欄はフィールドで、華氏ラベルと摂氏ラベルはラベ

ル、そして計算ボタンがボタンです。最初にこれらを作り出し、適切な位置に置きます。

計算ボタンの動作の中では、まず華氏入力欄から使っている人が打ち込んだものを取り出し、変数「華氏」に入れています。次に、上の公式を使って、摂氏の温度を計算し、変数「摂氏」に入れています。最後に、その摂氏の温度を摂氏出力欄に書いて表示させます。

ここで使っている変数「華氏」「摂氏」では入れる時に先頭に「：」をつけていませんが、それはこれらの変数はメソッドの外で読むことはないので、このオブジェクトの中だけの変数で構わないからです。

もっと具体的に言えば、メソッドの中で「：」なしの変数に書き込むと、その変数はその同じオブジェクトのすべてのメソッドで共通に使われる変数になります。これを**インスタンス変数**と呼びます。

また、プログラムを書くときにいちいち途中結果を変数に入れないで、次のように書くこともできます。

```
計算ボタン：動作＝「
  華氏＝（華氏入力欄！読む）。
  摂氏出力欄！（５＊（華氏－ ３２）／９）書く」。
```

ここでは変数「摂氏」を使わずに計算した結果を摂氏出力欄に直接書いていますが、さらに変数「華氏」を使わないで次のようにしてもよいです。

```
計算ボタン：動作＝「
  摂氏出力欄！（５＊（（華氏入力欄！読む）－ ３２）／９）書く」。
```

なのですが、プログラムが短くなっても読みにくくなると結局後で何をやっているのか分からなくなって困りますから、自分で適当と思う程度に簡潔に書くようにするのがよいでしょう。

演習 4 フィールド (入力欄)、ラベル、ボタンを使って、次のような値の計算を行うプログラムを作ってみなさい。

a. 例題とは逆に、摂氏の温度を華氏に変換する。ちなみに、変換の計算式は $F = 9 \times C \div 5 + 32$ になります。

b. 円の半径を入力して、面積を表示する。円周率は 3.1416 とすること。

c. 2 つの値を入力して、その合計を表示する。入力に使うフィールドが 2 つ必要なことに注意。

d. 次々と数を累計していく専用のプログラム。入力欄は 1 つで、現在の合計はラベルに表示させること。また、ボタンとして「加算」ボタンと現在の合計を 0 にする「クリア」ボタンの 2 つを用意すること。

3.4　計算して図形を描く

前の節では計算した結果は数値でしたが、図形を描く時に必要な値を計算することもあります。正多角形を描く例題を見てみましょう (図 3.5)。

```
カメ太＝タートル！作る　ペンなし　－８０　８０　位置　ペンあり。
説明文＝ラベル！『3 以上の整数を入れてください』作る。
入力欄＝フィールド！作る。
実行ボタン＝ボタン！『描く』作る。
実行ボタン：動作＝「
  辺数＝入力欄！読む。
  辺長＝６００／辺数。
  角度＝３６０／辺数。
 「カメ太！（辺長）歩く
   （角度）右回り」！（辺数）繰り返す　図形にする」。
```

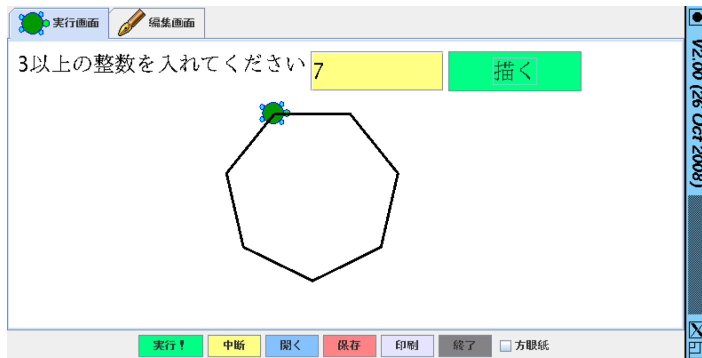

図 3.5: 正多角形を描くプログラム

タートルを作ったあと、ペンなし (線を引かない状態) にして位置を上の方に動かし、ペンあり (線を引く状態) に戻しています。GUI 部品の用意はこれまでやったのと同様で、あとはボタンの「動作」の中で正多角形を描く部分だけです。

具体的には入力された値 N に基づいて正 N 角形を描くわけですが、大きすぎて画面から出てしまわないように、辺の長さの合計を一定 (ここでは 600) にすることにしました。ですから、1 辺の長さは「$600/N$」を計算して決めます。また、曲がる角度は合計で 360 度 (1 周) なので、辺 1 つ描くごとに「$360/N$」度曲がることになります。これらが計算できれば、「1 辺の長さだけ進み、曲がる」動作を N 回繰り返すことで正 N 角形が描けるわけですね。

このように、プログラム中で計算に基づいて描くことで、規則的な (きれいな) 図形を描くことができるわけです。

演習 5 次のような図形を計算に基づいて描くプログラムを作ってみなさい。

a. 5 以上の素数 N を入力して「N 角星」を描く。1 辺の長さは適当に決めてよい。

b. 角度 D と繰り返し数 N を入力して「角度 D で折れ曲がった、線が N 本から成るジグザグ形」を描く。線 1 本の長さは適当に決めてよい。

c. 整数 N を入力して、1 辺 20 の正方形が 20 ずつ隙間を空けて N 個横に並んだものを描く。

d. 整数 N、M を入力して、1 辺 20 の正方形が 20 ずつ隙間を空けて横に N 個、縦に M、縦横に並んだものを描く。

第4章　ものを動かして制御する

この章では、ものを連続的に動かすことでアニメーションを作り出し、それをユーザに制御してもらうことで、簡単なゲームを作ってみましょう。そのためには、プログラムの中で条件を判断して、実行の流れを「枝分かれ」させる方法も知っておく必要がありますから、そちらから先に取り上げていきます。

4.1　枝分かれ

少し込み入ったプログラムを作ろうとすると、条件に応じて図 4.1 のように実行の流れを「枝分かれ」させる必要が出て来ることがあります。たとえば「数当てゲーム」なら、打ち込んでもらった数が正解の場合と不正解の場合とで違うメッセージを表示させたいですよね。というわけで、ここで「枝分かれ」の作り方を学んでおきましょう。

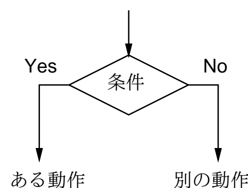

図 4.1: 枝分かれ

ドリトルでは、枝分かれは次の形で書き表します。これは、指定した条件が成り立っている (「はい」である) 時だけ、一連の動作を実行させるものです。

```
「…条件…」! なら「…条件成立時の動作…」実行。
```

ここに出て来る「…」はおなじみのブロックで、「なら」「実行」はメソッド名です。つまりドリトルでは、枝分かれにもブロックを活用するわけです。

条件としては色々なものが書けますが、とりあえず表 4.1 の比較が使えることを覚えておけばよいでしょう。いくつか注意しておいて欲しいことがあります。

- 「等しい」は「==」のように「=」を 2 つ続けて書く。これは、1 つだけの「=」を代入の意味で使ってしまっているからです。
- 「等しくない」「以下」「以上」は日本語では「≠」「≦」「≧」という文字がありますが、これらの文字がない英語版などでも使えるように「!=」「<=」「>=」という 2 文字の組み合わせでも表せるようになっています。

表 4.1: 比較演算子

名前	機能	サンプル
==	等しい	x==10
!=，≠	等しくない	x!=10，x≠10
<	より小さい	x<10
<=，≦	以下である	x<=10，x≦10
>	より大きい	x>10
>=，≧	以上である	x>=10，x≧10

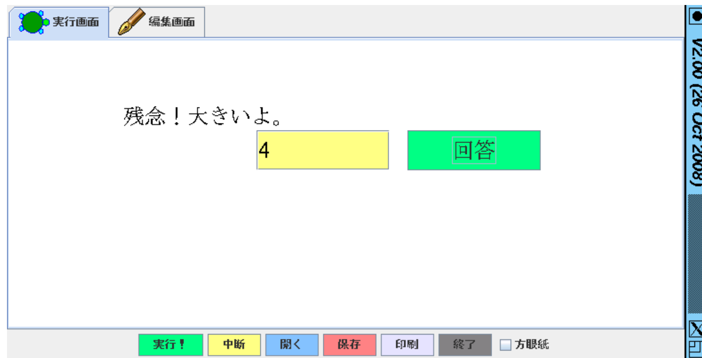

図 4.2: 数当てゲーム

ではいよいよ、枝分かれと比較を使って、数当てゲームを作ってみましょう (図 4.2)。

```
正解＝ random(９)。
表示欄＝ラベル！『数当て：１…９の数を予想してください』作る
　－２５０　９０　位置。
入力欄＝フィールド！作る　－１００　６０　位置。
回答ボタン＝ボタン！『回答』作る　７０　６０　位置。
回答ボタン：動作＝「
　回答＝（入力欄！読む）。
　「回答＝＝正解」！なら「表示欄！『正解です。』書く」実行。
　「回答＜正解」！なら「表示欄！『残念！小さいよ。』書く」実行。
　「回答＞正解」！なら「表示欄！『残念！大きいよ。』書く」実行」。
```

このプログラムは、実行開始するとまず、1〜9 までの数からランダムに 1 つ数を選び、変数「正解」に入れます。続いて、これまでと同様に表示欄、入力欄、回答ボタンを作ります。続いて回答ボタンが押された時の動作を設定していますが、この中で枝分かれを使うわけです。

動作の最初に、入力欄からユーザが打ち込んだ値を取り出し、変数「回答」に入れます。それからいよいよ、枝分かれの処理です。具体的な内容は見ての通り、回答と正解が等しければ「正解です」、回答が小さければ「小さい」、大きければ「大きい」と表示欄に入れるだけです。簡単でしたね。

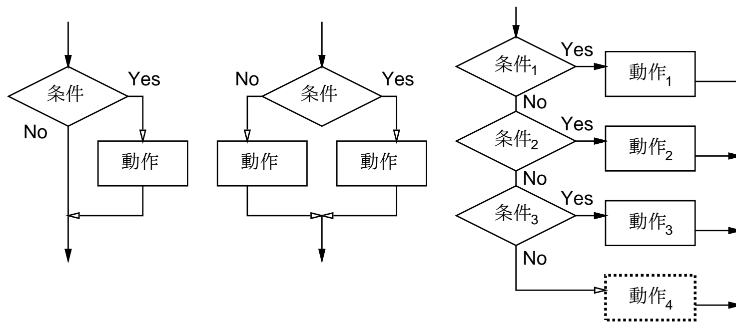

図 4.3: 枝分かれの実行の流れ

ところで、上で学んだ枝分かれは一番簡単なもので、条件が成り立った時「だけ」指定した動作を実行するものでした。これを図で示すと、図 4.3 左のようになります。ところで場合によっては、図 4.3 中のように、「条件が成り立った場合と成り立たなかった場合それぞれについて」別の動作を指定したいかもしれません。そのような場合は、次のようにブロックを追加してください (この「そうでなければ」というのもメソッド名です)。

```
「…条件…」！なら「…条件成立時の動作…」
　　そうでなければ「…条件不成立時の動作…」実行。
```

もう 1 つのよくある枝分かれの形として、図 4.3 右のように、いくつかの条件を順に調べて行き、条件が成り立った場合にそれぞに対応して決まった動

作をする、というものがあります。これを**連鎖形の枝分かれ**と呼びます。さらに点線のように、どの条件も成り立たなかった場合の動作を指定することもあります。これをドリトルで書く場合は次の形になります。[1]

```
「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4」実行。
```

プログラムを作っていて、条件による処理が複雑になったときは、これらの枝分かれが必要になるかも知れません。

演習 1 数当ての例題を次のように改良してみなさい。

a. 動作は変更しないで、連鎖型の枝分かれを使うように直してみなさい。(等しくも小さくもなければ大きいに決まっているので、条件は2つでよくなることに注意。)

b. 回答できる回数の上限が3回とし、3回で正解しなかったら正解を表示するようにしてみなさい。(注意! a. の改良の後でやった方がうまく行くはずです。)

c. 正解したり、回数リミットで終わった後で、再度ゲームを行うボタンを追加してみなさい。

演習 2 「丁半」を作ってみなさい。最初の持ち金を1000円とし、掛け金を指定して「丁」「半」のいずれかのボタンを押すとサイコロを振り、丁半が当たっていたら掛け金が倍、外れたら掛け金が没収になります。掛け金が10万円を超えるか無くなると終わりです。(ヒント: 丁半の確率は等しいので、`random(2)` が1か2かで丁半を判断するのが簡単でよいでしょう。)

4.2 タイマーとアニメーション

これまでの章では、タートルや図形を動かすと「一瞬で」動いてしまっていました。しかし、ゲームなどを作るときは、もっと「ゆっくり」「徐々に」動かしたいですね。それにはどうしたらよいでしょうか。

みなさんは、テレビ、映画、動画などは「パラパラまんが」のように、少しずつ変化する絵を短い時間間隔で次々に見せることで「動いている」ように見せていることをご存じだと思います (図4.4)。

ですから、プログラムで絵を描く時も、まず最初の絵を作ったあと、

[1]「!」が必要なのは最初の条件の直後だけなので注意してください。また、点線部分が不要なら、最後の『そうでなければ「動作4」』を取り除きます。

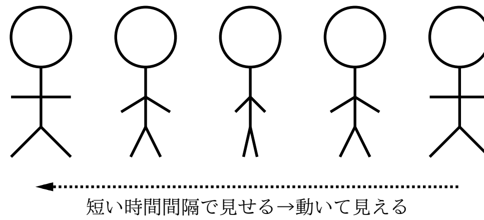

図 4.4: 動画の原理

少し待つ→絵を少し動かす→少し待つ→絵を少し動かす→少し待つ→…

という操作を繰り返して行くことで、絵を徐々に動かすことができます。これを**アニメーション**と言います (テレビアニメの「アニメ」もこれが縮まった言い方ですね)。

ドリトルでは、このように「少しの待ち時間をはさみながら動作を繰り返す」ためには**タイマー**オブジェクトを使用します。タイマーに対しては、次の 2 つを指定します。

- 間隔 — 待ち時間、つまり動作と動作の間の間隔 (指定しないと 0.1 秒)。
- 時間 — 動作を繰り返す時間の長さ (指定しないと 10 秒)。

または、時間の代わりに回数を指定してもいいです。たとえば、1 秒間隔で 60 回繰り返すというのは、1 秒間隔で 1 分間繰り返すのと同じことです。タイマーはタートルと同じように「作る」で作り出し、その後で「間隔」「時間」「回数」のメソッドで間隔や時間や回数を設定します。設定が終わったら、ブロックを指定して「実行」させることで、そのブロックに掛かれた動作を指定された時間間隔で繰り返し実行させてくれます。

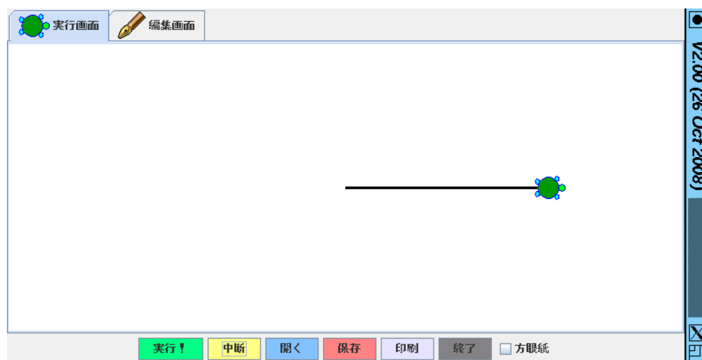

図 4.5: タートルをアニメーションで動かす

表 4.2: タイマーのメソッド

名前	機能	サンプル
作る	新しいタイマーを作る	時計＝タイマー！作る。
間隔	動作の間隔 (秒) 設定	時計！0.1 間隔。
時間	動作時間の長さ (秒) 設定	時計！10 時間。
回数	動作の起こる回数設定	時計！100 回数。
実行	タイマーを動作させる	時計！「…動作内容…」実行。
中断	タイマーの動作を途中であっても止める	時計！中断。
待つ	タイマーの動作終了 (または中断) を待つ	時計！待つ。

タイマーオブジェクトのメソッドを表 4.2 に示します。基本的には上で説明したように、間隔と時間 (または回数) を指定した後で「実行」を使います。なお、「待つ」を使うとタイマーの実行が完了するまで待つことができます。また、「中断」を使うと、指定した時間や回数に満たない時点でも、タイマーの実行をそこで終わらせることができます。

では実際にやってみましょう。次のプログラムは、0.1 秒間隔で 10 秒間タートルを徐々に前進させますが、その前進する「歩幅」が徐々に (時間とともに) 長くなっています。

```
歩幅＝１。
カメ太＝タートル！作る。
時計＝タイマー！作る 0.1 間隔 10 時間。
時計！「カメ太！（歩幅）歩く。歩幅＝歩幅＋ 0.1」実行。
```

アニメーションなので図だけ見てもあまりようすが分かりませんが、これを動かしているようすを図 4.5 に示します。

演習 3 例題では時間とともにだんだん動きが速くなっていましたが、これを逆に遅くなるようにしてみなさい。

演習 4 1 回歩くごとに一定角度曲がるようにすることで、うずまき型などの図形を描かせてみなさい。

4.3　衝突時の動作

さて、アニメーションができるようになったので、これを使って簡単なゲームを作ってみましょう。ここで作るゲームは「リフティング」つまり落ちて来るボール (タートル) をパドルではね上げ続けるものです。パドルは最初に

カメ太に太さ 20、長さ 100 の線を引かせて、これを図形にして使うことにします。また、カメ太は線は引かないようにして、下向きにして、最初は画面の上の方に配置します。

```
歩幅＝１。
カメ太＝タートル！作る　２０　線の太さ　１００　歩く。
パドル＝カメ太！図形にする　０　－１００　位置。
カメ太！９０　右回り　ペンなし　０　１００　位置。
カメ太：衝突＝「：歩幅＝０－　：歩幅」。
時計＝タイマー！作る 0.1 間隔 10 時間。
時計！「カメ太！（歩幅－　２）歩く。歩幅＝歩幅＋ 3」実行。
```

「カメ太：衝突＝…」というのは何でしょうか? これはカメ太の「衝突」というプロパティにブロックを入れています。つまりカメ太に「衝突」というメソッドを持たせているのです。何のためにでしょう?

タートルは、動いた結果他のタートルや図形と重なった状態になると、自分自身が持つ「衝突」というメソッドを呼び出すように作られています。ですから、「衝突」というメソッドを作って、そこにパドルに当たったときの動作を記述しているのです。

ここでは当たった時の動作は「歩幅をマイナスにする」です。たとえば歩幅が 10 だったら、それを－ 10 に書き換えます。歩幅がマイナスだと、カメ太は後退しますから、今まで落ちていたものが逆に上昇するようになり、つまり「パドルで反射させる」ことができるわけです (図 4.6)。

なお、反射のとき「マイナス 3 から」引いているのは、少し勢いを増やして反射させるように工夫したためです。また、「：歩幅」のように「：」がつけてあるのは、グローバル変数の歩幅を書き換えるためです。

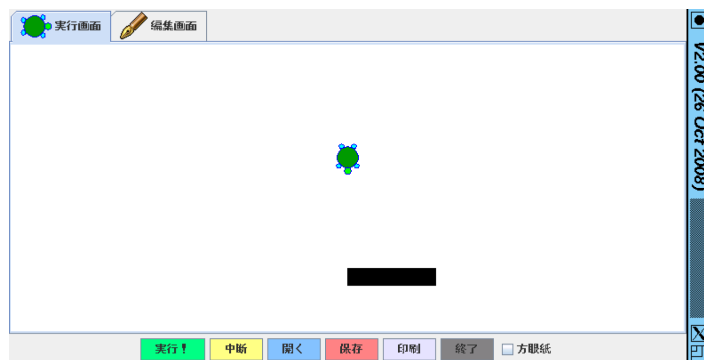

図 4.6: タートルをパドルで反射させる

```
歩幅＝１。
カメ太＝タートル！作る　２０　線の太さ　１００　歩く。
パドル＝カメ太！図形にする　０　－１００　位置。
カメ太！９０　右回り　ペンなし　０　１００　位置。
カメ太：衝突＝「：歩幅＝-　3-　歩幅」。
時計＝タイマー！作る 0.1 間隔 20 時間。
時計！「カメ太！(歩幅-　２）歩く。歩幅＝歩幅＋ 3」実行。
```

演習 5 例題を手直しして、跳ね返る高さが「だんだん高くなる」「だんだん低くなる」などの調節をしてみなさい。

4.4　ゲームに仕立てる

では材料が揃ったので、ボタンを使ってパドルを左右に動せるようにして、ボールが画面からとび出さずに 10 秒間リフティングを続けられたら成功と判定することで、ゲームにしてみましょう (図 4.7)。

図 4.7: リフティングのゲーム

変数として、「歩幅」を含めて次の変数を使うことにしました。プログラムを計画するときはこのように、どのような変数があって、それぞれにどのような情報を保持させているかを、あらかじめ整理しておくことが大切です。

- 歩幅 — ボールが縦方向にどれくらいの速さで下降/上昇しているか
- 水平 — ボールが水平方向にどれくらいの速さで移動しているか
- Ｐ — パドルの位置 (中央が０)
- 成功 — ボールが保持できていれば「はい」、画面からはみ出たら「いいえ」を入れる。

では、以下にプログラムを示してから順番に説明しましょう。

```
歩幅＝１。水平＝０。Ｐ＝０。成功＝はい。
左＝ボタン！『左』『LEFT』作る。
右＝ボタン！『右』『RIGHT』作る。
表示＝ラベル！『リフティング』作る。
左：動作＝「：Ｐ＝Ｐ－５。パドル！（Ｐ－５０）　－１００　位置」。
右：動作＝「：Ｐ＝Ｐ＋５。パドル！（Ｐ－５０）　－１００　位置」。
カメ太＝タートル！作る　２０　線の太さ　１００　歩く。
パドル＝カメ太！図形にする　（Ｐ－５０）　－１００　位置。
カメ太！９０　右回り　ペンなし　０　１００　位置。
カメ太：衝突＝「
　：歩幅＝－３－歩幅。ｘ＝カメ太！横の位置？。
　：水平＝水平＋0.3＊（ｘ－Ｐ）」。
時計＝タイマー！作る 0.1 間隔 20 時間。
時計！「
　「（カメ太！縦の位置？）＜－１１０」！なら
　　「：成功＝いいえ。時計！中断」そうでなければ
　「（カメ太！横の位置？）＜－３００」なら
　　「：成功＝いいえ。時計！中断」そうでなければ
　「（カメ太！横の位置？）＞３００」なら
　　「：成功＝いいえ。時計！中断」実行。
　カメ太！（水平）（０－歩幅）移動する。歩幅＝歩幅＋3」実行。
時計！待つ。
「成功」！なら
　「表示！『おめでとう！』書く」そうでなければ
　「表示！『残念！』書く」実行。
```

最初の行は４つの変数に最初の値を入れています。次に、パドルを左右に動かすための２つのボタンと、メッセージを表示するための表示欄 (ラベル) を用意しました。左右のボタンを作るときのパラメータに「LEFT」「RIGHT」という文字列を余分に渡していますが、これらを指定することで、ボタンをマウスでクリックする代わりに左矢印キーや右矢印キーを押しても同じ動作が起きるようになります。メッセージは最初は単に「リフティング」と表示しています。

パドルやボールの準備は先の例と同じです。衝突の動作については、先の例の動作に追加したものがあります。まずボールの横位置を変数ｘに取り出し、それとパドルの位置との差を取って 0.3 倍したものを、変数「水平」に加えています。こうすると、反射させるときにパドルの右側で打つとボールが左方向 (マイナス方向) に、左側で打つと右方向 (プラス方向) に動くようになり、よりゲームらしくなります。

時計で繰り返し実行する動作はだいぶ増えています。つまり、ボールが画面の下に行ってしまったとき、左に出てしまったとき、右に出てしまったときはいずれも、変数「成功」を「いいえ」に変更して、「中断」でタイマーの動作を止めます。その後はこれまで通りボールを動かしますが、今度は縦だけでなく横にも動かすので「歩く」ではなく「移動」を使い、「歩幅」は下向きの速さなので０から引いて向きを逆にします。「歩幅」を毎回３ずつ増やすのはこれまでと同じです。

最後に、時計に対して「待つ」を実行することで、ゲームの終了 (10 秒経過するか、または「中断」されるか) まで待ちます。その時点で変数「成功」の値に応じて、表示欄に「おめでとう！」か「残念！」のどちらかを書き込みます。

だいぶ長くなってきましたが、このように順番に必要なオブジェクトや動作を追加していくことで、ゲームのような込み入ったものでも組み立てて行けることが分かります。また、「画面からはみ出ているか」「成功しているか」などの条件判断も使う必要がありました。

ところで、「画面の下に出る」「左に出る」「右に出る」の 3 つの場合とも、その場合におこなう動作は同じものでした。このようなときは、「3 つの条件のどれかが成立していれば」のように条件をまとめて書いた方がわかりやすいかも知れません。。このような条件の組み合わせには「どれか」というオブジェクトを使います。

```
どれか！（条件）（条件）（条件）本当
```

これの類似品として、複数の条件について「そのすべての条件が成り立てば」という条件を作りたいときは「全部」というオブジェクトを使います。

```
全部！（条件）（条件）（条件）本当
```

これらの結果は真または偽なので、それを用いてこれまでと同様「なら」を使って枝分かれすればよいわけです。たとえば、「どれか」を使って時計の動作の部分を書き直すと次のようになるでしょう。

```
時計！「
  「どれか！((カメ太！縦の位置？）＜- １１０）
            ((カメ太！横の位置？）＜- ３００）
            ((カメ太！横の位置？）＞３００）本当」！なら
   「：成功＝いいえ。時計！中断」実行。
 カメ太！（水平）（０- 歩幅）移動する。歩幅＝歩幅＋ 3」実行。
```

演習 6 ゲームの難しさ (ボールの縦や横の移動の速さ) を調節してみなさい。また、「得点」が出るようにしなさい。得点としては、単にボールを反射させた回数にしてもいいですし、ボールを斜めに反射させた時の方が得点が高くなるようにしてもよいです。

第5章　プログラムの計画と設計

プログラムを作る時には、どのような画面や動き方にするのかを「計画」し、その計画を達成するためにはどのようなオブジェクトや機能を使い、どのような手順を使うのかを「設計」する必要があります。この章では、新しいゲームを「計画」「設計」してから作っていきます。また、その中で使う「条件を持つ繰り返し」と「配列」についても説明します。

5.1　プログラムの計画

プログラムを作る時には、まず「どのような画面で」「どのような動作をする」プログラムにするかを計画します。作りはじめてしまうと、途中から大きく計画を変更するのは大変ですから、最初に「作る価値があり」「自分の腕前で作れる」ようにうまく計画することが大切です。[1]

計画を立てるときは、漫然と考えていてもうまく行きません。紙などに画面の様子や動きかたを何通りかスケッチしてみる、などの方法が有効です。

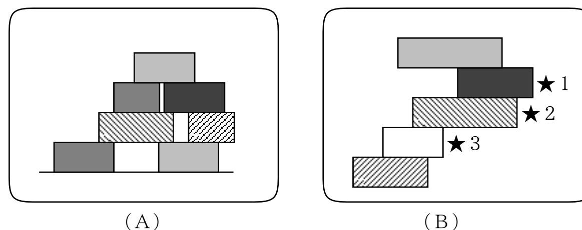

図 5.1: 「1 山崩し」の計画

ここでは「1 山崩し」という (仮に) 名前をつけたゲームを計画してみましょう。最初は、将棋の山崩しのように、沢山の駒が積んであって、それを取っていって「崩れた」ら負け、のようなゲームを作りたいと思ったとします (図 5.1A)。でも、「駒」は長方形で表すとしても、どのようになったら「崩れた」ことになるのかの判断が簡単ではなさそうですし、ちょっと作るのに難易度が高いように思えます。

[1] うまくプログラムが作れても、使って面白くなければしかたがありません。また、いくら完成したら面白いものでも、完成させられなければ意味がありません。ですから、計画のよしあしで、プログラムのよしあしの多くが決まってしまうのです。

そこで図 5.1B のように、(バランスとかは無視して) 縦にまっすぐ積んで、だるま落としみたいに途中の段を抜いて1段ぶんずつ低くしていくのではどうでしょう (一番上と一番下の2枚は抜けないことにします)。間違っていちどに2段以上低くしたら失敗です。2以上人で順番に抜いて行き、途中で失敗した人は抜けていき、最後に残った人 (または一番上と一番下の2枚だけにした人) が勝ちとします。たとえば図の場合、★1、★3、★2の順で取れば2枚にできますが、★2を先に抜いてしまうと失敗になります。これなら、ある段を抜いた時にその上と下の駒が重なっているかだけ調べれば成否が分かりますから、そんなに難しくなさそうです。

このように、「うまくできるかどうか」を考える時には、求めるものが条件としてうまく書き表せるかどうか、うまく計算できるかどうかも、併せて考えていくことになります。

演習1 自分で作ってみられそうなゲームプログラムの計画を立ててみなさい (本章の後の方まで読んでからやってもよい)。

5.2 条件を持つ繰り返し

駒を積み上げるのに必要なので、ここで「条件を持つ繰り返し」の説明をしましょう。

前章まででは繰り返しとして、「回数を指定した繰り返し」と「一定時間間隔での繰り返し」を扱いましたが、もう1つ重要な種類の繰り返しが「条件を持つ繰り返し」です。ここでいう条件とは、枝分かれのところで使った条件と同じものです。その書き方も、枝分かれとよく似ていて、次のようになります。

```
「…条件…」! の間「…繰り返される動作…」実行。
```

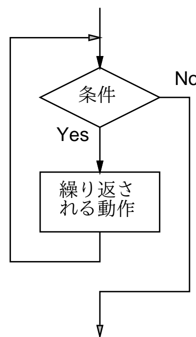

図 5.2: 条件に基づく繰り返し

これを実行の流れとして図で表すと、図 5.2 のようになります。そして、その実行のされ方をもっと具体的に見ると、次のようになるわけです。

・条件を調べる。
・成立していたら、「繰り返される動作」をおこなう。
・条件を調べる。
・成立していたら、「繰り返される動作」をおこなう。
…
・条件を調べる。
・成立していなかったら、繰り返しを終わる (先へ進む)。

では、実際に駒を積み上げるところまでを行うプログラムを見てみましょう。変数としては、次のものを使います。

- カメ太 — 長方形を描くためのタートル。
- ｈ — 次に駒を置く縦方向の位置。
- ｘ、ｗ — 直前に置いた駒の左端の位置と幅。
- 塗色、幅、高さ、ｐ — 次に置く駒の色と幅と高さと水平位置。
- 四角 — 新しく作成した駒 (長方形)

今回はタートルは四角を作るだけのために使うので、姿が見えず、線の太さも０に設定します。次に、駒を積み上げる一番下の位置をｈに設定し、最初は中央から積み始めるのでｘとｗを０に設定します。

その先が条件を持つ繰り返しで、繰り返しの中で駒を積むと、位置ｈがその分だけ大きい値になるので、画面の上端近く (位置 140) より低い間積み続ける、つまりそれを超えたら繰り返しを終わるようにしています。ちょっと先走って繰り返し部分の最後を見ると、「ｈ＝ｈ＋高さ。」がありますね。これでｈの値を増やしていくので、いずれは繰り返しが終わることが確認できます。[2]

```
カメ太＝タートル！作る　消える　０　線の太さ。
ｈ＝- １５０。ｘ＝０。ｗ＝０。
「ｈ＜１４０」！の間「
  塗色＝色！(random(255))(random(255))(random(255)) 作る。
  高さ＝ random(10) ＋１０。幅＝ random(40) ＋４０。
  ｐ＝ｘ- 幅＋１＋ random(幅＋ｗ- ２)。
  「カメ太！（幅）歩く　９０　左回り（高さ）歩く
      ９０　左回り」！２　繰り返す。
  四角＝カメ太！図形を作る（塗色）塗る（ｐ）（ｈ）位置。
  ｘ＝ｐ。ｗ＝幅。ｈ＝ｈ＋高さ。
」実行。
```

[2]このように、条件を指定する繰り返しでは、繰り返しを続けて行くと条件が成り立たなくなって繰り返しが終わるように作ってあることを確認してください。これに失敗すると、永遠に繰り返し続ける (そこから先に進まない) プログラムができてしまいますから。

では繰り返しの中で、駒を置くところを見ましょう。色はランダムに、また駒の幅と高さは 10〜20、40〜80 の範囲でランダムに生成します。では、置く位置はどすればいいでしょう。縦の位置は h ですが、横の位置 p は下にある駒に「引っかかる」ように積む必要があります。図 5.3 のように図を描いて考えると、上に積む駒の水平位置 p は一番左に寄った時でも「x − 幅＋ 1」である必要があり、またそこから右に「幅＋ w − 2」までは寄っても大丈夫ですから、右に寄るぶんを乱数で計算することにします。

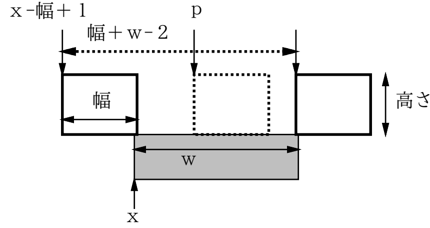

図 5.3: 「引っかかる」状態で積む

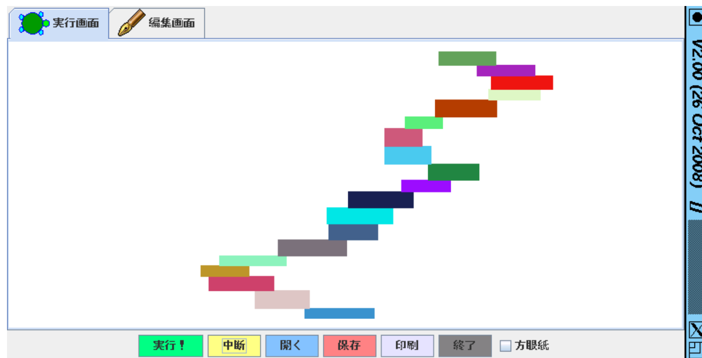

図 5.4: 乱数で駒を積む

位置が計算できたら、カメ太に長方形を描かせ、図形にして塗ってから計算した位置に置きます。そのあと、次の繰り返しに備えて x と w は今おいた駒の幅と高さを入れ、 h は高さぶんだけ増やします。プログラムを動かしたようすを図 5.4 に示します。

演習 2 長方形ではない別の形を積むようにしてみなさい。たとえば、三角形を積むなど。積み方も自分で工夫してみなさい。

5.3 配列とその扱い

上のプログラムでは、駒は置いてしまったらあとは画面に見えるだけで、その位置などはプログラムでは覚えていませんでした。しかし、「1 山崩し」のプログラムを作るためには、積み上げた駒を全部覚えておかないと、抜いた駒より上の駒を 1 段ぶんおろすことができません。このような場合を扱うための機能である「配列」について、ここで説明しておきましょう。

配列とは、複数のものを入れることができる「いれもの」です。配列の中に入っている個々のものを**要素**と呼びます。要素は配列を作る時に指定したり、後からメソッド「書く」で追加でき、メソッド「読む」で何番目かを指定して取り出すことができます。配列のメソッドを表 5.1 に示します。

表 5.1: 配列のメソッド

名前	機能	サンプル
作る	配列を生成する	並び＝配列！1　3　5　作る。
書く	配列に要素を追加	並び！7　9　書く。
上書き	指定番号の要素を差し替え	並び！1　『X』上書き。
位置で消す	指定番号の要素を削除	並び！2　位置で消す。
読む	指定番号の要素を取得	y＝並び！3　読む。
要素数？	配列中の要素の数を返す	N＝並び！要素数？。
結合	各要素を（区切り文字を挟み）連結した文字列を返す	s＝並び！『、』結合。
それぞれ実行	各要素をパラメタとしてブロックを繰り返し実行	並び！「|x|…x を処理…」それぞれ実行。

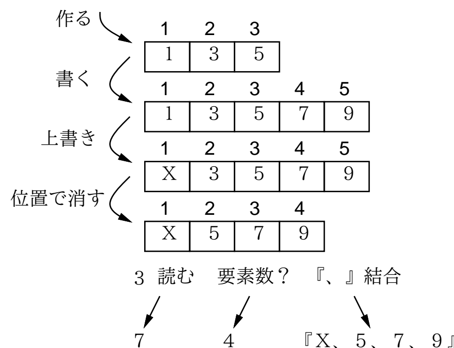

図 5.5: 配列の操作

図 5.5 は表 5.1 のサンプルを実行させた時のようすを示したものです。箱の中に入っているのがそれぞれの要素で、箱の上の数字はその箱が「何番目」

かを示します。また、表の最後にある「それぞれ実行」は新たな種類のループで、指定したブロックを配列の要素の数だけ繰り返し実行しますが、その時要素を順番にパラメタとして渡してくれます。ですから、配列の各要素に対して順番に何かの処理をする場合に便利です。

では、ちょっと寄り道をして、配列を使った簡単な「記憶ゲーム」を作ってみましょう (図 5.6)。これは 10 個の整数が 10 秒間表示されてから消えるので、消えた後で覚えている数を次々に回答し、いくつ思い出せるかで点数をつけるものです。

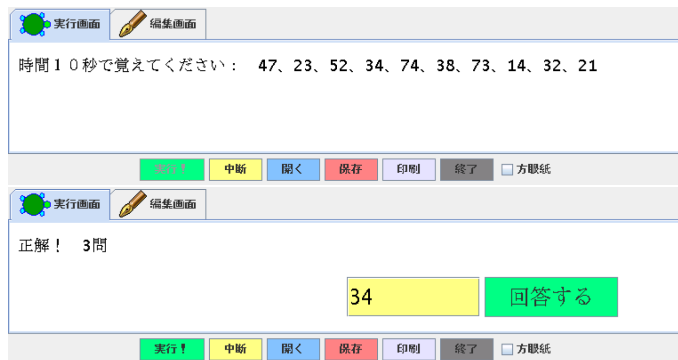

図 5.6: 数記憶ゲーム

```
並び＝配列！作る。正解数＝０。
並び：探す＝「｜ａ｜ ｉ＝-　１。ｋ＝１。
  自分！「｜ｘ｜
  「ｘ＝＝ａ」！なら「ｉ＝ｋ」実行。ｋ＝ｋ＋１」それぞれ実行。
  ｉ」。
「ｘ＝ random(100)。並び！(ｘ) 書く」！１０　繰り返す。
表示欄＝ラベル！(『時間１０秒で覚えてください：　』＋
                  （並び！『、』結合））作る　１８　文字サイズ。
時計＝タイマー！作る　１０　間隔　１　回数「」実行。時計！待つ。
表示欄！『では、数を思い出してください。』書く。
入力欄＝フィールド！作る　０　０　位置。
回答＝ボタン！『回答する』作る。
回答：動作＝「
  ｉ＝並び！（入力欄！読む）探す。
  「ｉ＝＝-　１」！なら
    「表示＝『残念！　』＋正解数+『問』」そうでなければ
    「正解数＝正解数＋１。表示＝『正解！　』＋正解数＋『問』。
      並び！(ｉ) 位置で消す」実行。
  「正解数＝＝１０」！なら「表示＝『全問正解！』」実行。
  表示欄！（表示）書く。
」。
```

まず、10 個の数は当然配列に覚えるので「並び」という変数に配列を入れ

ます。また、正解した数も記録したいので「正解数」という変数を用意しておきます。

次ですが、あとで入力された数が配列の何番目にあるか (または無いか) を調べたいので、この配列に「探す」というメソッドを追加しています。メソッドには何番目かを探したい値をパラメタとして渡します。変数 i が結果の番号で、最初は「見つからない」という印を表す− 1 を入れておきます。変数 k は番号を数えるためのもので、最初は 1 です。次に、この配列自身の[3]「それぞれ実行」を使って各要素を先頭から順番に調べます。この繰り返しの中では、取り出した要素が探したい値と等しければ k を i に入れることで番号を覚えます。その後、毎回 k を 1 増やすので、k は常に各要素の入っている位置を保持していることになります。最後に i とあるので、これがメソッドの結果として返されます。このように、ひとまとまりの仕事をメソッドとして用意しておくことで、プログラムの流れが見通しやすくなります。

この後が数記憶ゲームの本体になります。まず 1〜100 の間の乱数を生成して並びに追加することを 10 回繰り返します。これが「問題」になります。次に表示欄に問題を表示します。その後、タイマーで 10 秒間隔で「からっぽのブロック」を 1 回実行し、その終了を「待つ」ので、ここで 10 秒の時間待ちになります。その後表示欄を書き換え、回答用の入力欄とボタンを作ります。

あとはボタンを押した時の動作のメソッドだけです。そこではまず、入力された値が並びの何番目にあるか探します。ここで先に作ったメソッド「探す」を呼び出しています。結果が− 1 であれば「残念」そうでなければ「正解」を、現在の正解数とつなげて「表示」という変数に入れます (すぐに表示しないのは、まだ表示を変更する可能性があるからです)。その後、正解であれば正解数を 1 増やし、また並びから i 番目の要素を削除することで回答された数値を並びから消します。その後、正解数が 10 なら「表示」の値を「全問正解」にとりかえます。そして最後に「表示」の値を表示欄に表示するわけです。

このように、配列を使うことで、複数の値を保持しておいて、その中から探したり値を取り除いたり追加したりしながら、プログラムを実行していくことができます。

演習 3 数ではなくさまざまなものの名前を記憶するように直してみなさい。または、数そのものではなく「3 + 5」などの計算式を 10 秒間表示して、答えを入力させるようにしてみなさい。(ヒント: 問題に出すものの名前は予め十分な数を別の配列に入れておき、そこからランダムに 10 個選ばせるとよいでしょう。計算式の場合は乱数を 2 回つかって 2 つの値を生成し、表示は足し算の形の文字列で行い、覚えておく方は足した結果にするのがいいでしょう。)

---

[3]「自分」というのはこのメソッドを持っているオブジェクトを表す名前です。「並び」でもいいと思うかもしれませんが、「自分」にしておけば、このメソッドをほかの配列にもそのまま使うことができるので、このような場合は「自分」を使うのがよいのです。

5.4 プログラムの設計

前節のように「できあがった」プログラムを説明されても、なるほどできているなあとは思っても、どうすればこのように作れるのか納得が行かないかも知れません。実際には、本章の冒頭で説明したような「計画」と実際に作る作業との間に「設計」が必要です。

「設計」というのは、プログラムの各部分をどのように作れば全体として「計画」が達成されるかを考えながら作りやすい「部分」に分けたり、それぞれの「部分」を作れるようにするためには、どのようなデータやオブジェクトをどのように保持しておけばいいかを考えて決める作業のことです。

人間は一度に沢山のことを考えようとするとわけが分からなくなりますから、「部分」に分けたり「データやオブジェクト」に着目したりすることで、考える部分を小さく (ないし特定側面に) 限定し、その部分に絞ってよく検討できるようにするわけです。なお、「どのように分けるか」「どの側面に限定するか」なども「考えなければならないこと」に含まれます。

では実際に、「1 山崩し」のプログラムを設計してみましょう。まず、データについて次の方針を立てます。

- 配列を用意し、そこに積んである駒を積んである順に入れておく。

もともとの駒が順に積んであるので、それをそのまま同じ順で配列に入れておくことで、分かりやすく、プログラムが作りやすくなります。このような、プログラムの中でのデータの「かたち」のことを**データ構造**と呼びます。

次に「長方形をクリックすると消える」ことについて考えます。これは、その図形にメソッド「動作」を設定しておけば、クリックしたときにその動作が実行されますから、そこで行えばよいでしょう。

このとき、長方形を画面から消すのは「自分」を消せばいいので簡単ですが、消したものは配列からも取り除く必要があります。消すのには前節のように「何番目を消す」と言えばいいのですが、それにはその長方形が何番目か分かっている必要があります。その方法としては、次の 2 つがとりあえず思いつきます。

- 前節でやったように、何番目かを「探す」。
- 各図形オブジェクトにそれが何番目かを持たせておく。

どちらがいいかは、もう少し検討を進めてから決めましょう。

次に「消えた長方形の上の長方形がその分だけ下がる」ことについて考えます。「上にある」というのは「番号が大きい」ということですから、消した長方形の番号 (それは分かっているものとして) より大きい番号のものを移動すればいいでしょう。とすると、それぞれの長方形に番号を持たせておく方

が作りやすそうです。[4]「その分だけ」というのはどうでしょうか。これも、各長方形にその高さを持たせておけばすぐ分かります。

次に「アウトかどうか、クリア (最後の 2 駒だけになった) かどうか」を知ることも必要です。クリアは配列の要素数が 2 かどうか見ればいいので簡単ですね。アウトかどうかは、消した長方形の直上と直下の長方形が今度は直接上下になるわけですから、この 2 つの長方形の位置関係で分かるはずです。

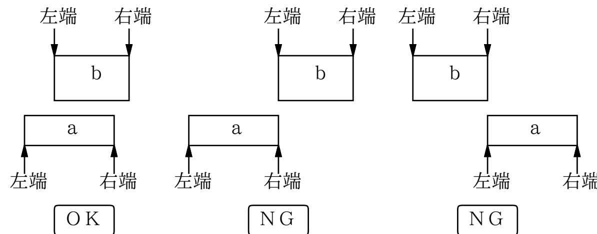

図 5.7: アウトかどうかの判定

図 5.7 のように図を描いてみると、上の長方形を a、下を b としたとき、「b の左端が a の右端より右」か「b の右端が a の左端より左」になったら NG で、そうでなければ OK と分かります。ですから、各長方形にその右端と左端の位置も持たせておけば判定は簡単です。

最初に長方形を積み上げるところは、既に作ってあります。本当はここも設計があるのですが、今回は紙面の都合で説明を省略します。

5.5 設計に基づくプログラムの作成

では設計に基づいて実際に開発したプログラムを見ていただきましょう。まず表示欄 (最初のメッセージを表示している状態) と配列を作ります。配列には「番号づけ」のメソッドを用意します。これは変数 i を 1 にしてから、「それぞれ実行」で要素を 1 つずつ取り上げ、そのオブジェクトの「番号」プロパティに i を設定してから i を 1 増やすことを繰り返すだけです。

その次ですが、「抜き出し」という名前でブロックを定義しています。このブロックは番号 i をパラメタとして受け取り、その番号の長方形を消してそれより上にあるものをその分だけ下に移し、さらにアウトかどうかの判定も行うという、ゲームのコアの部分を処理するようになっています。実際にはこのブロックは別の場所から「実行」で呼び出されるのですが、その場所に中身の処理を書くとごちゃごちゃになるので、予めここで定義しておいて呼

[4] 普通の言語だと「持たせておく」のが面倒なので別の方法を選ぶところかも知れません。ドリトルでは図形などのオブジェクトに自由にプロパティを追加していいので、「持たせておく」のが簡単なのです。

び出すようにしています。このように、プログラムを作成するときはその記述が整って読みやすいように配慮することも大切です。

ブロックの処理内容を見てみましょう。まず最初に並びからｉ番目の図形(消す図形) を取り出し、変数ａに入れます。そしてａは「消える」で見えなくし、続いて並びのすべての要素について、それの番号がｉより大きいなら(上にあるので)、ａの厚みのぶんだけ下に移動します。その後、並びからｉ番目の要素を取り除き、そうすると番号が変わるので並びの番号づけを呼び出して番号をつけ直します。ここまでで抜く処理は終わりました。次に、新たなｉ番目の要素 (消した長方形のすぐ上にあった長方形) とｉ－１番目の要素(消した長方形のすぐ下にあった長方形) を取り出し、変数ｂとａに入れます。そして、ａとｂが噛み合わないならアウト、そうでなくて図形の残り数が2ならクリア、それ以外は何番目の図形を抜いたかを、表示欄に表示します。

```
表示欄＝ラベル！『１段だけ落ちるように抜く駒を選択』作る。
並び＝配列！作る。
並び：番号づけ＝「ｉ＝１。
  自分！「｜ｘ｜ｘ：番号＝ｉ。ｉ＝ｉ＋１」それぞれ実行」。
抜き出し＝「｜ｉ｜ａ＝並び！（ｉ）読む。
  ａ！消える。
  並び！「｜ｘ｜
    「ｘ：番号＞ｉ」！なら「ｘ！０（０－ａ:厚み）移動する」実行。
  」それぞれ実行。
  並び！（ｉ）位置で消す。並び！番号づけ。
  ｂ＝並び！（ｉ）読む。ａ＝並び！（ｉ－１）読む。
  「どれか！(ａ:右端¡＝ｂ:左端)(ｂ:右端¡＝ａ:左端) 本当」！なら「
    表示欄！『アウト！』書く
  」そうでなければ「（並び！要素数？）＝＝２」なら「
    表示欄！『クリア！』書く
  」そうでなければ「
    表示欄！（ｉ＋『番を抜きました』）書く
  」実行」。
カメ太＝タートル！作る　消える　０　線の太さ。
ｈ＝－１５０。ｘ＝０。ｗ＝０。
「ｈ＜１４０」！の間「
  塗色＝色！(random(255))(random(255))(random(255)) 作る。
  高さ＝random(10) ＋１０。幅＝random(40) ＋４０。
  ｐ＝ｘ－幅＋１＋random(幅＋ｗ－２)。
  「カメ太！（幅）歩く　９０　左回り（高さ－１）歩く
      ９０　左回り」！２　繰り返す。
  四角＝カメ太！図形を作る（塗色）塗る（ｐ）（ｈ）位置。
  四角：左端＝ｐ。四角：右端＝ｐ＋幅。四角：厚み＝高さ。
  四角：動作＝「抜き出し！（自分：番号）実行」。
  並び！（四角）書く。ｘ＝ｐ。ｗ＝幅。ｈ＝ｈ＋高さ。
」実行。
（並び！１　読む）：動作＝「」。
（並び！（並び！要素数？）読む）:動作＝「」。並び！番号づけ。
```

この先の部分は基本的に前に出て来た積み上げるだけのプログラムと同じ構造です。ただし、四角を作るごとに、新たに次の処理を行っています。

- 長方形の左端、右端の位置と高さ (厚み) を図形オブジェクトのプロパティとして格納する。
- 長方形にメソッド「動作」を定義する。
- 作った長方形を配列に追加する。

各駒をクリックすると、そのメソッド「動作」が呼ばれます。その中でやていることは、自分の番号をパラメタとして、先に作った抜き出しのブロックを呼び出すだけで、あとはブロックの方で既に説明した抜き出しの処理をすべてやってくれます。

なお、一番上と下の駒は取り除けないので、1 番と N 番 (N は並びの要素数) の図形については、メソッド「動作」を空っぽのブロックに書き換えています。最後に、各図形が正しい番号を持ってから始まる必要があるので、並びに対して定義したメソッド「番号づけ」を呼び出します。これで全部です。動かしているようすを図 5.8 に示します。

図 5.8: 完成した「1 山崩し」

このように、設計に従って部分ごとにプログラムを考えて組み合わせていくことで、最初は込み入った難しい処理に思えても、少しずつ作成していって最終的に完成させることができるのです。

以上で、この本の内容はおしまいです。ここから先は皆様が、自分でやりたいことを考えて、プログラムを作っていってください。

演習 4 先に計画した自分独自のゲームを設計し、制作してみなさい。

索 引

アニメーション, 41
インスタンス変数, 33
オブジェクト, 11
オブジェクト指向言語, 11
コード, 9
コンピュータ, 5
スライダー, 30
ソフトウェア, 5
タートル, 12
タートルグラフィクス, 11
タイマー, 41
データ構造, 54
ドリトル, 8
パラメータ, 13
パラメタ, 24
ブロック, 16
プログラミング言語, 8
プログラム, 5
ボタン, 27
メソッド, 11

繰り返し, 15

原点, 9

座標, 9

実行画面, 9

処理系, 8

色オブジェクト, 21
図形オブジェクト, 17

配列, 51

文字列, 28
変数, 20
編集画面, 9

要素, 51
乱数, 25

連鎖形の枝分かれ, 40

GUI, 27
GUI 部品, 27