計算機プログラミングI 2005　久野クラス　＃10

久野　靖*

2006.1.6

はじめに

あけましておめでとうございます。授業でも言いましたが、2006 年度からはこの「計算機プログラミングI」という科目はなくなり、「情報科学」というクラス指定の必修科目としてアルゴリズム/データ構造などを中心に扱う科目になります。なので、このクラスのように私が自由に題材を選択して皆様に「面白いプログラム」ができるように誘導するというのは今年度が最後になるわけです。あとちょっとですが、面白いことはまだまだありますので頑張りましょう。今回は前半で「音」を取り上げ、併せて抽象データ型について学ぶことにします。後半は継承の利用方法についてのお話ですが、もう「次回までの課題」はないので、後半のは話だけ聞いておいて頂ければよいことにします。

1　計算機によるサウンドの扱いと JavaSound

ここまで数週にわたり、Java のさまざまな「目を楽しませる」機能、具体的にはユーザインタフェース、グラフィクス、イメージ、アニメーションなどを見て来ました。

今回は、視覚に続くもう 1 つの感覚である聴覚、つまり音を扱います。Java 2 の枠内で音を扱うには、単なるサウンドファイルの再生機能 (古くからある方法) と、さまざまな音の加工を可能にした JavaSound とがありますが、前者はちっとも面白くないので今回は後者のみ、その中で特に MIDI 関係を扱います。音関係のすべての例題は JDK 1.3.x 以降が必要です。

今回の主題に入る前に、音とは何か、そして計算機による音の扱いではどんなことが問題になるかについて、簡単に説明しておきましょう。まず音とは、誰もが理科の時間に学ぶように、「空気の振動」です。つまり、空気の圧力が大きくなったり小さくなったりする「波」が次々に伝わって行くことで、音が伝わるわけです。

一番単純な波を考えると、その特徴は振動の幅つまり「最大と最少の圧力の差」と、振動の頻度つまり「1 秒間に何回大小の間を往復するか」と、で表されます。前者はもちろん、音の大きさ (強さ) に対応します。そして後者は「周波数」とも呼ばれ、単位は Hz(ヘルツ) で表します。周波数が大きいほど「高い調子の音」になります。

音は空気の振動だと言いましたが、テレビやラジオやオーディオ機器の中ではどうでしょう? これらの機器の中では、音の振動を (マイクロフォンなどによって) 電圧の大小に変換し、電気信号として扱いますね。ですから、時とともに電圧が変化するようすで音の情報が表せるのです。計算機の中でもこのような電気信号による音を扱います。

もう 1 つ、計算機で重要なのは、計算機が扱うデータはすべてディジタルデータだということです。しかし音は連続した振動によって表されています。ですから、音を計算機に取り込むところでは連続した (アナログの) 情報をディジタル情報に変換すること (AD 変換) が必要ですし、逆にスピーカーを通して人間に聴けるようにするにはデイジタル情報をアナログに戻す (DA 変換) 必要があります。この点は、計算機でなくても CD、MD などのディジタルオーディオ機器であれば同様です。

AD 変換の原理は、連続した信号の値を非常に短い一定時間間隔ごとに計測し、その計測値をディジタル化された数値 (黒丸で表しました) に直すというものです。ここで、計測の頻度 (縦線の細かさ) を「サンプリング周波数」といい 8KHz から 44KHz くらいの値が使われます。周波数が高いほど元のアナログ信号に忠実なディジタルデータとなります (その代り時間あたりのデータ量が大きくなります)。

また、それぞれの計測値をどれくらいの細かさ、すなわち何ビットの数値として表すかも選択できます (横線の細かさに対応)。こちらは「サンプルサイズ」と呼ばれ、具体的にはだいたい 8 ビットか 16 ビットのどちらかです (もちろん 16

*筑波大学大学院経営システム科学専攻

ビットの方がより忠実なデータになります)。ただし、データ量を減らすために、目盛りのつけ方を線形にする代りに大きい音のところは粗くする方法 (μ- law と a-law の 2 通りがあります) が使われることもあります。線形の場合は「リニア PCM」(PCM は Pulse Coded Modulation の意味) と呼ばれ、さらに数値の表現が符号つきと符号なしの 2 通りあります。そして 16 ビットの数値の場合は記録データ上で上位の桁のバイトが先 (上位優先、ビッグエンディアン) か下位の桁のバイトが先 (下位優先、リトルエンディアン) かの違いもあります。そしてステレオでは左右チャンネルのデータが必要なのでデータ量は 2 倍になります。

やれやれ、と思ったところですみませんが、まだあります。計算機で「音楽」を扱う場合は、上のようなサンプリングに基づく方式の他に、「どんな楽器の、どの音階を、どんな強さで、どれだけ鳴らす」という情報を連ねた形で扱うこともできます。このようなものを MIDI 形式のデータと呼びます。

サンプル形式と MIDI 形式の違いは、ちょうとお絵描きツールのペイント形式 (絵を色の着いた多数の点の集まりで表す) とドロー形式 (絵を直線や円などの幾何学図形の組み合わせで表す) の違いだと思ってもらえればよいでしょう。サンプル形式では (その解像度の範囲内で) 任意の音や音楽が表せますが、データ量が多く、また「どこにどんな音があるか」などを調べたり加工するのは大変です。一方、MIDI 形式では表せるのは基本的に「楽譜で記述できるような音楽」に限られますが、データ量は比較的小さく、楽器を取り換えたり速さや調子を変更したりといった加工も容易です。

MIDI 形式のデータをサウンドデータに変換して再生するのは、シンセサイザーとかシーケンサと呼ばれる機器の仕事でしたが、その中身はデータ処理ですから計算機で実行可能です。最近の計算機では CPU 能力が高くなっているため、そのようなソフトウェア (ソフトウェアシンセサイザー) も実用的になっています。

最後に、実際に音を取り込んだり再生するためにはもちろん、計算機にサウンドデバイスが備わっている必要もあります。こちらは、最近の計算機では備わっているものがほとんどでしょう。もちろん iMac にも備わっていますので、それで今回これを使おうというわけです。

Java2 以前の Java では上述の通り、音の機能は単にファイルを再生できるというものに過ぎませんでした。しかし、世の中には音を録音したり加工・再生する多くのソフトが存在しており、Java でもこれらのソフトと同様の機能が自在にプログラムできるようにしたい、というのが多くの人の希望でした。

このため、Sun は Java で音を扱う標準的な API の開発を進め、JavaSound と名付けました。JavaSound は最初は別途配付のパッケージ群でしたが、JDK 1.3 からは JDK に標準で付属するようになっています。以下では JavaSound の基本的な概念から始めて、それでどのようなことができるかを一通り見て行くことにします。なお、JavaSound の Web サイトは次のところにありますから、詳しく知りたい人はこちらも参照してみてください。[1]

```
http://java.sun.com/products/java-media/sound/
```

JavaSound で通常使われるクラス群は次の 2 つのパッケージに含まれています。

- `javax.sound.sampled` — サンプル形式のサウンドデータを扱う。
- `javax.sound.midi` — MIDI 形式のサウンドデータを扱う。

やっぱり「音楽」を直接扱う方が分かりやすくて楽しいので、以下では MIDI 形式の話に絞って説明していきます。

2 javax.sound.midi パッケージ

2.1 MIDI の基本概念

MIDI とは Musical Instrument Digital Interface の頭文字を取ったもので、もともとは電子楽器どうしや電子楽器と計算機を相互に接続するための信号線の規格でした。たとえば、ミュージシャンがキーボード (鍵盤) を演奏すると、その信号が線を伝わってシンセサイザー (Synthesizer、音を合成する装置) に入り、そこでさまざまな音色の音が発生してアンプで増幅され、スピーカーから再生される、といった風景はどこでも見かけますが、どこのメーカーのキーボードをどのシンセサイザーにつなぐこともできます。これは MIDI がコネクタの形や信号の伝送方法の標準を定めてくれているからなのです。この部分を「MIDI ワイアプロトコル」と言います。

たとえば、最も基本的な音を出す/止める指示では、音の高さをノートナンバーと呼ばれる 0〜127 の範囲の整数で指定します。ノートナンバー 60 が中央の「ド」に相当し、1 増える/減るごとに半音ずつ高く/低くなります。また、音を

[1] このほかに、JavaSound に新しい機能を追加する開発者が使うためのパッケージとして javax.sound.sampled.spi と javax.sound.midi.spi がありますが (SPI は Service Provider Interface の略)、ここでは扱いません。

出す時にはベロシティ(Velocity、速度) も 0〜127 の範囲の整数で指定しますが、これは鍵盤を叩いたり弦を弾いたりする「強さ」に対応します。

さて、このような標準ができると、次は人間が鍵盤で演奏しなくても、あらかじめ記憶させた信号を正しいタイミングでシンセサイザーに送り、自動的に演奏を行わせる装置が作られるようになりました。これをシーケンサ (Sequencer) と呼びます。シーケンサには鍵盤その他の電子楽器から信号を入力させて記憶させることもできますし、(当然) 計算機から同様の情報を読み込ませることもできます。というわけで、ワイアプロトコルに曲の情報を保存するために必要な機能が追加されて、今日のような MIDI 形式のデータができあがりました。

MIDI のデータに含まれている各種の情報は、対応する `javax.sound.midi` パッケージのクラスやインタフェースによって表されています。以下では主要な情報の種類をこれらのクラス/インタフェースと対応させながら説明して行きます。紙面の関係から細かい説明はできませんので、適宜 API ドキュメントと照らし合わせて読んでください。

2.2 メッセージとイベント

MIDI では楽器や機器に対する指令を「メッセージ」と呼んでいます。メッセージは次の 3 種類があります。

- `ShortMessage` — 最も基本的なメッセージで、音を出す/止める、楽器 (プログラム) を変更するなどの機能があります。
- `SysexMessage` — システム固有 (SYStem EXclusive) メッセージ。各楽器メーカなどが提供する固有の機能を使うためのもの。
- `MetaMessage` — 演奏の速さや歌詞などのメタ情報 (シーケンサや計算機むけの情報) を格納するためのメッセージ。

なお、これら 3 つのクラスは共通の親クラス `MidiMessage` のサブクラスになっています。

さて、これらのメッセージは特定のタイミングでシンセサイザーに送られる必要がありますが、この「ある時点に送られること」を MIDI ではイベントと呼びます (AWT などで使って来たイベントとよく似ていますね)。MIDI のイベントはその名も `MidiEvent` クラスによって表されます。個々の `MidiEvent` のインスタンスは MIDI メッセージとそれが送られるタイミング (tick) の情報を対にしたものです。

tick とは曲のタイミングを表すための「きざみの最小単位」だと思ってください。tick の細かさは MIDI データを作るときに選べますが、その指定方法として次の 2 種類があります。

- SMPTE タイムコード基準 — SMPTE(Society of Motion Picture and Television Engineers、動画・テレビ技術者協会) が定めた標準規格では、1 秒間当たりの映像のコマ (フレーム) 数として 24、25、30、29.97 の 4 種類があるので、そのどれを使用し、さらに 1 フレームをいくつの tick に分割するかを指定します。この指定方法では 1 つの tick がどれだけの長さかは絶対的に定まります。
- PPQ 指定 — PPQ は Pulse Per Quater-Note(4 分音符あたりのパルス数) の意味で、4 分音符 1 つの長さをいくつの tick に分割するかだけを指定します。

具体的な値としては、たとえば PPQ として 384 を指定したりします。えらく細かいと思うでしょうが、MIDI では tick より小さいタイミングは表せませんから、音のタイミングを細かく制御するためにはある程度細かくする必要があるわけです。[2]

2.3 シーケンスとトラック

MIDI でひとまとまりの曲や演奏を表す単位をシーケンスと呼び、midi パッケージでは `Sequence` クラスによって表されます。tick の長さは 1 つのシーケンスでは 1 つに固定なので、`Sequence` オブジェクトのコンストラクタでは上で述べたどの指定方法で tick をどれくらい細かく取るかを指定する必要があります。

1 つのシーケンスには 1 つ以上のトラックが含まれます。たとえばスタジオ用のマルチトラックレコーダでは 1 つのトラックにボーカル、1 つのトラックにバックコーラス、他のトラックに各楽器を 1 つずつ録音するといった使い方をしますが、MIDI のトラックも同様に使うことができます。トラックは Track クラスによって表されます。`Sequence` クラス

[2] それにしても何で 384 なのでしょう? 音楽では 8 分音符、16 分音符、32 分音符と音符を半々に短くして行くので 2 のべき乗が便利ですが、3 連符もありますから 3 の倍数にもなっていた方が便利です。そこで 128×3 で 384 としたわけです。

のメソッド createTrack() でシーケンスにトラックを追加でき、また getTrack() で現在そのシーケンスに含まれるすべてのトラックを Track オブジェクトの配列として取得できます。

Track オブジェクトに対しては、メソッド add() で MidiEvent オブジェクトを追加したり、get() で N 番目の MidiEvent オブジェクトを取り出すなどの操作が行えます。

なお、MIDI データを格納するファイル形式のうち、MIDI Type 0 と呼ばれる形式ではトラックを 1 つだけしか格納できません。これに対し、MIDI Type 1 形式であれば任意個数のトラックを格納できます。MIDI 機器によっては MIDI Type 0 しか読み込めないものがありますが、全部の MidiEvent を 1 つのトラックにまとめるのでは不便なので、通常は Type 1 を使っておいて、必要なら Type 0 に変換するようなプログラム (フリーソフトもあります) を使えばよいでしょう。

2.4 チャネルとプログラム

MIDI ではチャネルとはシンセサイザーが持つ 1 つの「音発生装置」に対応しています。ですから、たとえばトランペット、ギター、ベース、ドラムセットで演奏する曲であれば、この 4 つをそれぞれ 1 つのチャネルに割り当てればよいわけです。なお、1 つのチャネルは同時に複数の音を生成できますから、和音を作り出すことができます。チャネルは MidiChannel オブジェクトで表されていますが、MIDI メッセージでは簡単のため、0 から始まる整数のチャネル番号で表します。

それぞれのチャネルは一時には一つの楽器に対応させられますが、MIDI ではこれをプログラムと呼びます。プログラムも 0〜127 の数値で表します。では、プログラム番号の何番がどの楽器に対応するのでしょう? 実は MIDI が始まったころにはこれらの番号に対する取り決めがなく、あるシンセサイザーではピアノで演奏されたメロディパートが、別のシンセサイザーだと全く別の楽器になってしまう、ということが起こっていました。これではあまりに不便なので、GM(General MIDI、汎用 MIDI) と呼ばれる標準が制定され、プログラム番号の何番がどの楽器かの割り当てを定めるようになりました。表 1 にその対応表を示します。

なお、各プログラム (楽器) は前述のノートナンバーを指定することでさまざまな高さの音を出せますが、その楽器に相応しくない高さの音は指定しても適切に演奏される保証がないことは注意してください。中央の 1 オクターブちょっと分のノートナンバーを掲載しておきます。

60	62	64	65	67	69	71	72	74	76
ド	レ	ミ	ファ	ソ	ラ	シ	ド	レ	ミ

もちろん、ノートナンバーの間は鍵盤で間に黒鍵のあるところ (全音) は 2、ないところ (半音) は 1 だけ違っています。

なお、リズムセクションはこれとは別で、チャネル 9 番に固定的に割り当てられており、どの打楽器かを**ノートナンバーで**指定します。この対応関係を表 2 に示しておきます。[3]

なお、GM は最小限の共通部分を定めたもので、機器メーカーによってはもっと多様な表現を可能にするように GS、GX など独自の拡張規格を定めています。

JavaSound では SoundBank インタフェースと Instrument クラスを通じて新しい楽器の音をロードして使用することもできます。1 つの SoundBank オブジェクトは複数の楽器の音情報を含み、ファイルから読み込んで来ることができます。SoundBank オブジェクトに含まれている個々の楽器の音情報が Instrument オブジェクトで表され、それらを個別にシンセサイザーにロードしたり、不要な楽器を除外したり、楽器のプログラム番号を割り当て直すこともできます。

2.5 MidiSystem と MIDI 機器

midi パッケージのクラス MidiSystem には、そのマシンに備わっている MIDI 機器群の情報を取得したり MIDI ファイルや SoundBank ファイルの読み書きを行うためのクラスメソッド群が含まれています。MIDI 機器はインタフェース MidiDevice で表されています。

MIDI 機器の代表は既に説明したシンセサイザーとシーケンサで、それぞれ Synthesizer インタフェース、Sequencer インタフェース (これらのインタフェースは MidiDevice のサブインタフェースです) によって表されています。システムが持っている標準のシンセサイザーとシーケンサはそれぞれクラスメソッド MidiSystem.getSynthesizer()、MidiSystem.getSequencer() で取得できます。これらは具体的なハードウェアである場合もありますし、ソフトウェア

[3] ここではチャネル番号やプログラム番号を 0 から始まる数値で表記していますが、一般向けの MIDI の解説書では 1 から数える方が普通です。そのような本では「リズムセクションはチャネル 10 に割り当てられている」となっているので、注意してください。

表 1: 表 1: GM の楽器番号

Piano	Chromatic percussion	Organ	Guitar
0 Grand Piano	8 Celesta	16 Organ1	24 Nylon-Str Gtr
1 Bright Piano	9 Glockenspiel	17 Organ2	25 Steel-Str Gtr
2 Electric Grand	10 Music Box	18 Organ3	26 Jazz Gtr
3 Honky-Tonk Piano	11 Vibraphone	19 Church Organ	27 Clean Gtr
4 E. Piano1	12 Marimba	20 Reed Organ	28 Muted Gtr
5 E. Piano2	13 Xylophone	21 Accordian	29 Overdrive Gtr
6 Harpsichord	14 Tubular Bell	22 Harmonica	30 Distortion Gtr
7 Clavinet	15 Dulcimer	23 Bandoneon	31 Gtr. Harmonics
Bass	**Strings and orchestral**	**Ensemble**	**Brass**
32 Acoustic Bass	40 Violin	48 String Ensemble	56 Trumpet
33 Fingered Bass	41 Viola	49 Slow Strings	57 Trombone
34 Picked Bass	42 Cello	50 Synth Strings1	58 Tuba
35 Fretless	43 Contrabass	51 Synth Strings2	59 Muted Trumpet
36 Slap Bass1	44 Tremolo	52 Choir	60 French Horn
37 Slap Bass2	45 Pizzicato	53 Voice	61 Brass Ensemble
38 Syn.Bass1	46 Harp	54 SynVox	62 Synth Brass1
39 Syn.Bass2	47 Timpani	55 Orchestra Hit	63 Synth Brass2
Reed	**Pipe**	**Synth lead**	**Synth pad**
64 Soprano Sax	72 Piccolo	80 Square Wave	88 Fantasia
65 Alto Sax	73 Flute	81 Sawtooth Wave	89 Warm Pad
66 Tenor Sax	74 Recorder	82 Syn.Calliope	90 PolySynth
67 Baritone Sax	75 Pan Pipes	83 Chiffer Lead	91 Space Vox
68 Oboe	76 Bottle Blow	84 Charang	92 Bowed Glass
69 English Horn	77 Shakuhachi	85 Solo Vox	93 Metal Pad
70 Bassoon	78 Whistle	86 5ths Saw Wave	94 Halo Pad
71 Clarinet	79 Ocarina	87 Bass & Lead	95 Sweep Pad
Synth sound effects	**Ethnic**	**Percussive**	**Sound Effects**
96 Ice Rain	104 Sitar	112 Tinker Bell	120 Fret Noise
97 Soundtrack	105 Banjo	113 Agogo	121 Breath Noise
98 Crystal	106 Shamisen	114 Steel Drum	122 Seashore
99 Atmosphere	107 Koto	115 Woodblock	123 Birdsong
100 Brightness	108 Kalimba	116 Taiko	124 Telephone
101 Goblin	109 Bagpipes	117 Melodic Toms	125 Helicopter
102 Echo Drops	110 Fiddle	118 Syn. Drums	126 Applause
103 Star Theme	111 Shanai	119 Reverse Cymbal	127 Gunshot

表 2: 打楽器とノートナンバーの対応

35 Acoustic bass drum	50 High tom	65 High timbale	80 Mute triangle
36 Bass drum 1	51 Ride cymbal 1	66 Low timbale	81 Open triangle
37 Side stick	52 Chinese cymbal	67 High agogo	
38 Acoustic snare	53 Ride bell	68 Low agogo	
39 Hand clap	54 Tambourine	69 Cabasa	
40 Electric snare	55 Splash cymbal	70 Maracas	
41 Low floor tom	56 Cowbell	71 Short whistle	
42 Closed hi-hat	57 Crash cymbal 2	72 Long whistle	
43 High floor tom	58 Vibraslap	73 Short guiro	
44 Pedal hi-hat	59 Ride cymbal 2	74 Long guiro	
45 Low tom	60 Hi bongo	75 Claves	
46 Open hi-hat	61 Low bongo	76 Hi wood block	
47 Low-mid tom	62 Mute hi conga	77 Low wood block	
48 Hi-mid tom	63 Open hi conga	78 Mute cuica	
49 Crash cymbal 1	64 Low conga	79 Open cuica	

的に実現されている場合もあります。1 つの機器 (ないしソフトウェアモジュール) がシンセサイザーとシーケンサの両方の機能を備えている場合もあります。

このほか、キーボードや電子楽器など MIDI インタフェースによって計算機に接続されているものも MIDI 機器です。MIDI 機器は MIDI 信号を送り出したり受け取ったりできますが、その出力コネクタは `Transmitter` インタフェース、入力コネクタは Receiver インタフェースで表されます。`MidiDevice` インタフェースが定義しているメソッド `getTransmitter()`、`getReceiver()` を用いて各 MIDI 機器のコネクタを取得でき、また `Transmitter` インタフェースが定義しているメソッド `setReceiver()` をもちいて出力コネクタに入力コネクタを「差し込んで」配線することができます。

だいぶ説明が長くなりました。ここから先は、具体的な例題プログラムを用いて midi パッケージの主要な機能を学んで行きましょう。

3 例題: シンセサイザーを使う

「音の出口」にいちばん近いシンセサイザーを扱う例題を示します。これは単独窓とレイアウトマネージャを使っているので今回は概要だけ説明します。後でこれらの概念について学んだら、窓の作り方、部品の配置などの部分まで含めて読めるようになるでしょう。今回はとりあえず、MIDI 関係の動作だけ見てくれればよいです。

このプログラムの機能を整理すると、上端にあるドロップダウンリストでチャネル番号、スライダでプログラム (楽器) 番号を選択し、鳴らしたい音に対応するノートナンバーのチェックをつけます。横に並んでいるスライダで速度 (強さ) を調節することもできます。そして「Sound」ボタンを押すとチェックされている音が同時に鳴ります。

このプログラムの基本方針としては、いちいち MIDI メッセージを用意する代りに `Synthesyzer` オブジェクトから取得した `MidiChannel` オブジェクトが持っているメソッド `noteOn()`、`noteOff()`、`progChange()` などを直接呼び出すことで音を鳴らしたり楽器を切り替えたりします。

ではコードを見て行きましょう。まず選択できるノートナンバーの範囲を定数で定義します。次に `Synthesizer`、`MidiChannel`、`Soundbank`、`Instrument` を入れる変数を用意します (チャネルや楽器は複数ありますから配列です)。その後は GUI 部品で、ボタン用のパネル、チャネル選択コンボボックス、プログラム選択スライダ、プログラム表示ラベル、「Sound」ボタン、中央のパネル、その中に入れるノートナンバーごとの部品、状態表示ラベルを変数に入れています。ノートナンバーごとの部品は後でクラス `NotePanel` として定義しています。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.sound.midi.*;

public class R10Sample1 extends JFrame {
  final int minNote = 35, maxNote = 81;
  Synthesizer synth;
  MidiChannel[] chan;
  Soundbank sbank;
  Instrument[] instruments;
  JPanel buttonPanel = new JPanel();
  int chanNum = 0;
  JComboBox chanSelect = new JComboBox();
  JSlider progSelect = new JSlider(0, 127, 0);
  JLabel progName = new JLabel("0 Grand Piano");
  JButton sound = new JButton("Sound");
  JPanel midPanel = new JPanel();
  NotePanel[] notes = new NotePanel[maxNote+1];
  JLabel statusLabel = new JLabel("status here...");
```

コンストラクタではまず窓のサイズと閉じる際の動作を指定し、次にシンセサイザーを取得して `open()` します。続いてチャネルの配列と `Soundbank` オブジェクトを取り寄せ、`Soundbank` オブジェクトからは `Instrument` の配列を取得しま

す。チャネル選択コンボボックスは取得したチャネルの個数ぶんの選択肢を 0 から順に用意します。続いて GUI 部品の配置に入り、レイアウトマネージャを設定してからボタンパネル、中央パネル、状態表示ラベルを入れます。

```
public R10Sample1() {
  setSize(600, 600); setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  try {
    synth = MidiSystem.getSynthesizer(); synth.open();
    chan = synth.getChannels(); sbank = synth.getDefaultSoundbank();
    if(sbank != null) instruments = sbank.getInstruments();
  } catch(Exception e) { statusLabel.setText(e.toString()); }
  for(int i = 0; i < chan.length; ++i) chanSelect.addItem(""+i);
  getContentPane().setLayout(new BorderLayout());
  getContentPane().add(buttonPanel, BorderLayout.NORTH);
  getContentPane().add(midPanel);
  getContentPane().add(statusLabel, BorderLayout.SOUTH);
```

チャネル選択コンボボックスが切り替わった時は変数 chanNum にそれを記録するとともに、そのチャネルを現在選択中の楽器に切り替えます。

```
  buttonPanel.add(chanSelect);
  chanSelect.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent evt) {
      chanNum = chanSelect.getSelectedIndex();
      chan[chanNum].programChange(progSelect.getValue());
    }
  });
```

プログラム選択スライダが動いた時は chanNum で示されている現在のチャネルのプログラムを切り替えるとともに、新しく選択されたプログラム番号と楽器を表す文字列をつなげたものを表示ラベルに設定します (配列 instruments がうまく初期設定できていない場合は番号だけの表示になります)。

```
  buttonPanel.add(progSelect); buttonPanel.add(progName);
  progName.setPreferredSize(new Dimension(150, 30));
  progSelect.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent evt) {
      chan[chanNum].programChange(progSelect.getValue());
      int prog = progSelect.getValue();
      if(instruments != null)
        progName.setText(prog + " " + instruments[prog].getName());
      else
        progName.setText("Program" + prog);
    }
  });
```

「Sound」ボタン上でのマウスの押し/離しではそれぞれ、チェックがついているノートナンバーを鳴らしたり止めたりします。

```
  buttonPanel.add(sound);
  sound.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent evt) {
      for(int i = minNote; i <= maxNote; ++i)
        if(notes[i].checked()) notes[i].noteOn();
    }
```

```
        public void mouseReleased(MouseEvent evt) {
          for(int i = minNote; i <= maxNote; ++i)
            if(notes[i].checked()) notes[i].noteOff();
        }
      });
```

ノートナンバーごとの部品 NotePanel は窓の中央部に GridLayout で規則的に配置します。2 列かつ数字が大きい方が上なのでループを使って計算しながら 2 個ずつ NotePanel を生成して配置し、なおかつ NotePanel オブジェクトは配列 notes に格納して行きます。音の範囲が奇数なので同じノートナンバーの部品を 2 回作らないように if 文でチェックしています。

```
      int d = (maxNote-minNote) / 2;
      midPanel.setLayout(new GridLayout(d+1, 2));
      for(int i = maxNote; i >= minNote+d; --i) {
        midPanel.add(notes[i-d] = new NotePanel(i-d));
        if(notes[i] == null) midPanel.add(notes[i] = new NotePanel(i));
      }
    }
```

main() メソッドは最初に動き出して窓を作るだけです。さて最後に入れ子クラス NotePanel を見てみましょう。このクラスは全体として中央パネルに入れたいので JPanel のサブクラスになっています。インスタンス変数として、ノートナンバー、チェックボックス、スライダを持ちます。コンストラクタではノートナンバーを受け取って記憶し、チェックボックスとスライダを自分自身の中に貼りつけます。チェックボックスの状態が変化したときはチェックの ON/OFF に対応して自分のメソッド noteOn()、noteOff() を呼びます。これらのメソッドは現在選択されているチャネルの noteOn()、noteOff() を (現在スライダに設定されている速度をパラメタとして) 呼び出すだけです。メソッド checked() は単にチェックボックスがチェックされているかどうかを返します。

```
    public static void main(String[] args) {
      (new R10Sample1()).setVisible(true);
    }
    class NotePanel extends JPanel {
      int note;
      JCheckBox check;
      JSlider velocity = new JSlider(1, 127, 100);
      public NotePanel(int n) {
        note = n; check = new JCheckBox(""+n); add(check); add(velocity);
        check.addItemListener(new ItemListener() {
          public void itemStateChanged(ItemEvent evt) {
            if(evt.getStateChange() == ItemEvent.SELECTED) {
              noteOn();
            } else if(evt.getStateChange() == ItemEvent.DESELECTED) {
              noteOff();
            }
          }
        });
      }
      public void noteOn() { chan[chanNum].noteOn(note,velocity.getValue());}
      public void noteOff() { chan[chanNum].noteOff(note,velocity.getValue());}
      public boolean checked() { return check.isSelected(); }
    }
  }
```

演習 1 このプログラムは Unix コマンド

```
cp  ~kuno/work/R10Sample1.java  R10Sample1.java
```

でコピーして来られるように用意したので、持って来て動かせ。どの楽器がどのような音を出すのか体験してみよ。また、チャネル 9 番を選ぶとプログラム番号と関係なく各ノートナンバーがそれぞれ異なる打楽器になることも確認してみよ。

このようにシンセサイザーを直接使うことで、たとえば「ゲームを実行しながらその場面に合った効果音を出す」といったプログラムも比較的簡単に作ることができます。

4 抽象データ型

4.1 例題: MIDI データを生成してみる

シンセサイザーの機能や音の出かたが分かったところで、いよいよ MIDI シーケンサを使って音楽を演奏してみましょう。このためには、シンセサイザに加えてシーケンサも取り出し、シーケンサの出口をシンセサイザの入口に設定します。それから音楽シーケンスを PPQ 型で作り、そこにメロディを用意します。メロディ自体は後述する自前のクラスで、「a(音程, 長さ)」というインスタンスメソッドで音を次々に追加して行けるようにしています (この音と次の音を重ねて出す場合は「a(音程, 長さ)」、音を出さない休符の場合は「s(長さ)」)。これらのメソッドは返値として自分自身を返すので、「a(...).a(...).a(...)...」のように次々と音を加えて行くのに好都合です (もちろんそのように作ったわけです)。

```
  import javax.sound.midi.*;

  public class R10Sample2 {
    static final int Q = 96;
    public static void main(String[] args) throws Exception {
      Sequencer seqr = MidiSystem.getSequencer(); seqr.open();
      Synthesizer synth = MidiSystem.getSynthesizer(); synth.open();
      seqr.getTransmitter().setReceiver(synth.getReceiver());
      Sequence seq = new Sequence(Sequence.PPQ, Q);
      Melody m1 = new Melody(100);
      m1.a(60,Q).a(62,Q).a(64,Q*2);
      m1.a(60,Q).a(62,Q).a(64,Q*2);
      m1.a(67,Q).a(64,Q).a(62,Q).a(60,Q).a(62,Q).a(64,Q).a(62,Q*2);
      m1.apply(seq.createTrack(), 65, 0, 95); //Alto Sax
//1   Melody m2 = new Melody(100);
//1   m2.p(60,Q*3).p(64,Q*3).a(64,Q*3).s(Q);
//1   m2.p(60,Q*3).p(64,Q*3).a(64,Q*3).s(Q);
//1   m2.p(60,Q*3).p(64,Q*3).a(64,Q*3).s(Q);
//1   m2.p(59,Q*3).p(62,Q*3).a(67,Q*3).s(Q);
//1   m2.apply(seq.createTrack(), 16, 0, 50); //Organ1
//2   new Melody(1).s(Q*2).concat(m1).raise(-24)
//2     .apply(seq.createTrack(), 32, 0, 60); //Acoustic Bass
      seqr.setTempoInBPM(160f); seqr.setSequence(seq); seqr.start();
      while(seqr.isRunning()) Thread.sleep(100);
      System.exit(0);
    }
```

そして、このメロディを楽器、チャネル番号、強さを指定してシーケンスに追加します (ここでは追加のつど新しいトラックを作っていますが、トラック数には限りがあるので複雑な音楽にするときは 1 つのトラックにもっと沢山の音を入れた方がいいかも知れません)。シーケンスができたら、テンポを指定してシーケンサにシーケンスをセットし、演奏開始します。そこですぐ終わってしまうと音が出ませんから、「シーケンサがまだ演奏中だったら、100 ミリ秒待つ」というループを入れて終わるのを待ってから System.exit(0) を実行しています。

ところで、単音の演奏だとつまらないので、コメントアウトの部分その 1 でもう 1 つメロディを作り、「ドミソ」「シレソ」の和音を出しています。ここは p(...)、s(...) のサンプルになっています。

また、新しくメロディを作らずに既にあるものを「加工」して使うこともできます。コメントアウトの部分その 2 では、まず「2 分音符ぶんの無音メロディを作り、それとさっきのメロディを連結したメロディを作り、そのメロディを 2 オクターブ下げたメロディを作り、それをベースに演奏させる」ようにしています。

では、補助クラス Melody を見てみましょう。基本的には「音」と「その音の開始時刻」を配列に入れて保持するだけですが、MIDI では「音」には終了時刻も必要なので、1 音につき配列を 2 要素ずつ使い、配列 note にはノートナン

バー (2 要素とも同じもの)、配列 `time` には時刻 (開始時刻と終了時刻) を入れています。なお、無音はマイナスのノートナンバーで表すことにしています。

```
    static class Melody {
      int note[], t0[], t1[], atime = 0, count = 0;
      public Melody(int n) { note=new int[n]; t0=new int[n]; t1=new int[n]; }
      private void add1(int n, int l) {
        if(count < note.length) {
          note[count] = n; t0[count] = atime; t1[count++] = atime = atime+l;
        }
      }
      public Melody a(int n, int l) { add1(n, l); return this; }
      public Melody p(int n, int l) { add1(n, l); atime -= l; return this; }
      public Melody s(int l) { add1(-9999, l); return this; }
      private void msg1(Track trk, int cmd, int ch, int v1, int v2, int t) {
        try {
          ShortMessage msg = new ShortMessage();
          msg.setMessage(cmd, ch, v1, v2); trk.add(new MidiEvent(msg, t));
        } catch(Exception e) { }
      }
      public void apply(Track trk, int pg, int ch, int v) {
        msg1(trk, ShortMessage.PROGRAM_CHANGE, ch, pg, 0, 0);
        for(int i = 0; i < count; ++i)
          if(note[i] >= 0) {
            msg1(trk, ShortMessage.NOTE_ON, ch, note[i], v, t0[i]);
            msg1(trk, ShortMessage.NOTE_OFF, ch, note[i], v, t1[i]);
          }
      }
//    public Melody raise(int d) {
//      Melody m = new Melody(count);
//      for(int i=0; i<count; ++i) m.add1(note[i]+d, t1[i]-t0[i]);
//      return m;
//    }
//    public Melody concat(Melody n) {
//      Melody m = new Melody(count + n.count);
//      for(int i=0; i<count; ++i) m.add1(note[i], t1[i]-t0[i]);
//      for(int i=0; i<n.count; ++i) m.add1(n.note[i], n.t1[i]-n.t0[i]);
//      return m;
//    }
    }
  }
```

private なメソッドというのは、外部から呼ぶことができず、このクラス内だけの下請け用メソッドになります。ここでは 1 音 (ないし無音) ぶんのデータを追加する add1()、MIDI のショートメッセージをトラックに追加する msg1() があります。後者では MIDI デバイス関係のエラーを無視するのに try-cache を使っています。

外部から音を追加するのに使う a()、p()、s() はパラメタを指定して add1() を呼び、最後に this を返すだけです (p()) は add1() が進めた時刻 atime を戻すことで次の音が同じ時刻に重ねて始まるようにしています)。

トラックにメロディをセットする apply() ではまず最初に楽器 (プログラム) 変更のメッセージを入れ、あとは各音ごとに「音開始」「音終了」のメッセージを対で生成して入れていきます。

演習 2 このプログラムをそのまま打ち込んで動かせ。ただしコメントアウト部分はパスしてよい。動いたらコメントアウトその 1(和音の部分) は入れてみた方がよい。その後さらに次のように改造してみよ。

a. 楽器やテンポを変更してみる。

b. メロディを変更したり、別のメロディを重ねてみる。

c. リズムセクションをつけてみる。(ヒント: 簡単にはトラック 9 番を使うだけ。きちんとするならそれ用のクラスを作るのがいいかも。)

4.2 抽象データ型

ところで、main() 側のコメントアウトその 2 についてどう思いますか? これを見ると、メロディオブジェクトというのは次のような性質がある (というか、そうなるように作られている) ことが分かります。

- あるメロディをもとにそれを加工したメロディを作り出す「演算」や、複数のメロディを組み合わせる「演算」が用意されている。

これはつまり、私達が普段使っている「1」「2」などの数値が「+」などの演算、「$sin(x)$」などの演算を持っているのとよく似ていますね。つまり、オブジェクトによって (メロディという) 一種の「データの種別」が作り出されていて、それを演算で組み合わせて行くことでさまざまなものが作り出せるわけです。このような考え方を、数や文字などの普通のデータ型と対比して**抽象データ型** (Abstract Data Types、ADT) と呼びます。つまり、内部的には複雑な構造を持っているのだけれど、外から見たらそのようなことは気にせず (抽象的に扱い)、さまざまな演算で自由に操作できる、というわけです。

演習 3 コメントアウトした残りの部分も打ち込み、メロディの「ずらし」や「連結」ができることを確認せよ。それができたら、さらに次のような「演算」(メソッド) を追加して、それを活用してメロディをさまざまに加工してみよ。

a. メロディを指定した回数反復したメロディを返すメソッド repeat(回数)。(ヒント: concat() を呼び出して利用すれば簡単。)

b. メロディの演奏時間を変更したメロディを返すメソッド scale(倍率)。たとえば 0.5 倍すると倍の速さで演奏される。(ヒント: concat() の前半と同様だが、ただし音の長さに倍率を掛ける。整数にキャストすることに注意。)

c. メロディを前後逆にしたメロディを生成して返すメソッド「reverse()」を作る。(ヒント: concat() の前半と同様だが、後ろから順に add1() すればよい。)

d. メロディの高低を指定し音 b を基準に反転する (たとえば「ドレミレ」を「レ」を基準に反転すると「ミレドレ」になる) メソッド inverse(b)。(ヒント: raise() 等と同様だが、無音以外の各音を b を基準にして計算し直す。)

e. その他、面白い加工をする「演算」を考えて作れ。

演習 4 何でも好きな音楽 (音) を生成するプログラムを作れ。

5 継承とインタフェースによるクラス階層設計

5.1 継承による差分プログラミング

さて、皆様もだいぶ大きなプログラムを書くように (おおむね :-) なったと思うので、プログラムが大きくなった時そのクラスの集まりをどういう構造にするか、という問題を取り上げておこう。1 つの考え方として、継承を駆使して、既にあるクラスを土台に新しいクラスを次々作って行くやり方がある。これを「差分プログラミング」とよぶ。例を見てみよう。まず冒頭部分はこれまで使って来たようなインスタンス変数類の準備。

```
import java.awt.*;
import javax.swing.*;

public class R10Sample3 extends JApplet {
  Image buf;
  boolean go;
  double time;
  Animation[] a = new Animation[20];
  int count = 0;
```

init() での初期化は配列 a にさまざまなオブジェクトを入れるだけ。ここでは 2 番目以降はコメントアウトしてある (あとで順次復活させる)。

```
  public void init() {
    a[count++] = new MovingCircle(Color.red, 100, 100, -30, 40, 15);
//  a[count++] = new MovingJack(new Color(80, 120, 180), 10, 10, 80, 30, 30);
//  a[count++] = new LightOnOffJack(new Color(40, 120, 80),
//                   10, 30, 20, 20, 30, Color.red);
//  a[count++] = new MovingSnowman(new Color(60, 100, 80),
//                   110, 30, 30, 20, 30, Color.red, 1.2);
//  a[count++] = new WavingSnowman(new Color(120, 240, 80),
//                   150, 70, 40, 20, 30, Color.red, 1.2);
//  a[count++] = new LongNeckSnowman(new Color(150, 80, 80),
//                   110, 70, 40, -15, 30, Color.red, 1.2);
//  a[count++] = new RotateSnowman(new Color(110, 180, 120),
//                   110, 110, -10, -15, 30, Color.red, 1.2);
  }
```

さて、アニメーションの開始/停止や描画関係はこれまでと同じなので特に説明不要 (Runnable インタフェースを実装するクラスを内部クラスの形で書いている)。

```
  public void update(Graphics g) {
    if(buf == null) { buf = createImage(getWidth(), getHeight()); }
    Graphics2D g2 = (Graphics2D)buf.getGraphics();
    g2.clearRect(0, 0, getWidth(), getHeight()); paint(g2);
    g.drawImage(buf, 0, 0, this);
  }
  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    for(int i = 0; i < count; ++i) a[i].draw(g2);
  }
  public void start() {
    go = true; time = 0.001 * System.currentTimeMillis();
    new Thread(new Runnable() {
      public void run() {
        while(go) {
          try { Thread.sleep(50); } catch(Exception ex) { }
          double dt = System.currentTimeMillis()*0.001 - time;
          for(int i = 0; i < count; ++i) a[i].addTime(dt);
          time += dt; repaint();
        }
      }
    }).start();
  }
  public void stop() { go = false; }
```

インタフェース Animation も同じ (というか前回よりも簡単)。

```
  interface Animation {
    public void draw(Graphics g);
    public void addTime(double dt);
  }
```

さて、ここから各図形クラスがはじまる。最初は何回もやった「飛ぶ円」。なお、このクラスで implements Animation を指定しているので、この子クラスもすべて Animation インタフェースに従うことになる (つまりインタフェースの implements 関係も継承される) ことに注意。

```
  static class MovingCircle implements Animation {
    Color cl;
    double gx, gy, vx, vy, rad;
    public MovingCircle(Color cl1, double x, double y,
                        double vx1, double vy1, double rad1) {
      cl = cl1; gx = x; gy = y; vx = vx1; vy = vy1; rad = rad1;
    }
    public void draw(Graphics g) {
      g.setColor(cl);
      g.fillOval((int)(gx-rad),(int)(gy-rad),(int)(rad*2),(int)(rad*2));
    }
    public void addTime(double dt) {
      gx += vx * dt; gy += vy * dt;
      if (gx<0.0 && vx < 0.0) { vx = -vx;}
      if (gx>400.0 && vx > 0.0) { vx = -vx;}
      if (gy<0.0 && vy < 0.0) { vy = -vy;}
```

```
    if (gy>300.0 && vy > 0.0) { vy = -vy;}
  }
}
```

ここから差分プログラミングになる。クラスの継承関係をぱっと見て読み取るのが難しいので、図を描いてみた (図 1)。

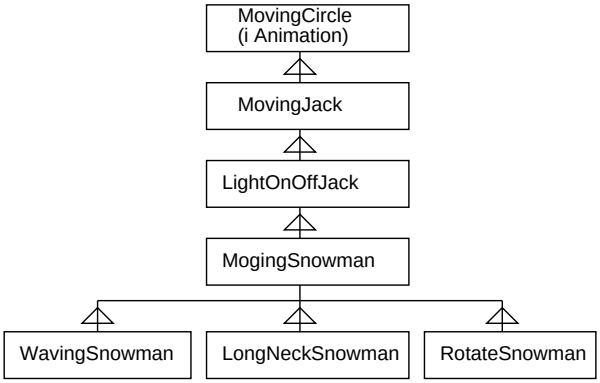

図 1: 差分プログラミングの例のクラス図

ではまず、飛ぶ円の中に目を描いて飛ぶ顔に直す。そのために、MovingCircle のサブクラスを作って draw() をオーバライドする。また、コンストラクタは継承できないのでこれも新しく作る。コンストラクタの中で「super(...)」により、親クラスのコンストラクタを呼び出していること、draw() の中で「super.draw(...)」により親クラスの draw() を呼び出していることに注意。このような、サブクラスの中から適宜親クラスの機能を参照できるような仕組みをマスターしておかないと、継承を使いこなすのはむずかしい。

```
static class MovingJack extends MovingCircle {
  Color eyeColor, mouthColor;
  public MovingJack(Color c, double x, double y,
                    double vx1, double vy1, double rad1) {
    super(c, x, y, vx1, vy1, rad1); eyeColor = mouthColor = c.brighter();
  }
  public void draw(Graphics g) {
    int u = (int)rad/4;
    super.draw(g);
    g.setColor(eyeColor);
    g.fillPolygon(new int[]{(int)gx-3*u, (int)gx-2*u, (int)gx-u},
                  new int[]{(int)gy-u, (int)gy-2*u, (int)gy-u}, 3);
    g.fillPolygon(new int[]{(int)gx+3*u, (int)gx+2*u, (int)gx+u},
                  new int[]{(int)gy-u, (int)gy-2*u, (int)gy-u}, 3);
    g.setColor(mouthColor);
    g.fillPolygon(new int[]{(int)gx-u, (int)gx, (int)gx+u},
                  new int[]{(int)gy+u, (int)gy+2*u, (int)gy+u}, 3);
  }
}
```

さて、次は目の色が時間とともに変わる顔を作る。このクラスは先のクラス MovingJack のサブクラスとし、addTime() を差し替えて時間の累計を取るようにした (その後で親クラスの addTime() も呼ぶ)。また draw() も差し替えて、現在の累計した秒が奇数か偶数かで eyeColor を切替えてから親クラスの draw() を呼ぶ。実は MovingJack で eyeColor を別の変数として保持していたのがポイントで、そうしてなければこう簡単に目だけ色を変えることはできない。このように、継承では親クラスが子クラスで書き換えるように変数の構成をうまく設計すること、子クラスはその親クラスの内部構造をわかった上で変数などを書き換えることが必要になる。

```
static class LightOnOffJack extends MovingJack {
  double atime = 0.0;
  Color lightOnColor;
  public LightOnOffJack(Color c, double x, double y, double vx, double vy,
                        double rad1, Color c1) {
    super(c, x, y, vx, vy, rad1); lightOnColor = c1;
  }
  public void addTime(double dt) {
    atime += dt; super.addTime(dt);
  }
  public void draw(Graphics g) {
    if((int)atime % 2 == 0) eyeColor = cl.brighter();
```

```
      else eyeColor = lightOnColor;
      super.draw(g);
    }
  }
```

次に、顔だけではつまらないので「本体」もつけて雪ダルマにしよう。これも先の LightOnOffJack のサブクラスで、「本体」をどれくらい顔から離すかを変数 dx、dy に保持することにした (もちろん、さらにサブクラスを作る時にこれらを活用する)。

```
  static class MovingSnowman extends LightOnOffJack {
    double ratio, dx = 0, dy = 0;
    public MovingSnowman(Color c, double x, double y, double vx1, double vy1,
                         double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1); ratio = r; dy = rad*2;
    }
    public void draw(Graphics g) {
      g.setColor(cl);
      g.fillOval((int)(gx+dx-rad*ratio), (int)(gy+dy-rad*ratio),
                 (int)(rad*ratio*2), (int)(rad*ratio*2));
      super.draw(g);
    }
  }
```

さて、雪だるまの本体を上下に振動させてみよう。それには、MovingSnowman のサブクラスを作って、時間を累計するところで dy の値を時刻の sin 関数に従って振動させればよい。

```
  static class WavingSnowman extends MovingSnowman {
    public WavingSnowman(Color c, double x, double y, double vx1, double vy1,
                         double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1, r);
    }
    public void addTime(double dt) {
      super.addTime(dt); dy = rad*2 - (ratio-1)*rad*(1+Math.sin(20*atime));
    }
  }
```

もっと別のいじくり方として、ろくろ首を作ってみよう。今度のクラスもまた MovingSnowman のサブクラスとして、今度は dy をのこぎり状に変化させ、なおかつ「首」の描画を追加している。ちょっとキタナイが、もともとの dy の値を壊さないために、super.draw() の直前で dy を増やし、その後 dy を戻している。

```
  static class LongNeckSnowman extends MovingSnowman {
    public LongNeckSnowman(Color c, double x, double y, double vx1,
             double vy1, double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1, r);
    }
    public void draw(Graphics g) {
      int len = (int)(rad*(atime%1));
      g.setColor(cl);
      g.fillRect((int)(gx-0.3*rad),(int)gy,(int)(0.6*rad),(int)rad*2+len);
      dy += len; super.draw(g); dy -= len;
    }
  }
```

また別の MovingSnowman のサブクラスとして、本体が顔のまわりを円周運動するというのも作ってみた。もう十分わかったでしょう?

```
  static  class RotateSnowman extends MovingSnowman {
    public RotateSnowman(Color c, double x, double y, double vx1, double vy1,
                         double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1, r);
    }
    public void addTime(double dt) {
      super.addTime(dt);
      dx = rad*2*Math.cos(atime); dy = rad*2*Math.sin(atime);
    }
  }
}
```

演習 5 上のプログラムもコピーできるように用意したので、コピーしてきてそのまま動かせ。動いたらどれかのクラスのサブクラスを作って新しい絵 (?) を追加してみよ。

5.2 複合 (コンポジション) による構造化

さて、差分プログラミングを見てどう思いましたか? 「なるほど、うまくやっているなあ」と思った人はだまされている。だって、この調子でつぎ足しつぎ足しして本当にきれいなプログラムができるとは思えないでしょう? たとえば「顔が四角いろくろ首」を作ろうと思ったら、ちょっとサブクラスを作って、というわけには行かない。一般に、N 個の選択肢と M 個の選択肢があった場合、その任意の組合せは $M \times N$ 通りになる。これをサブクラスでやろうとして $M \times N$ 個のサブクラスを作っていては、とても手に負えない。

これに対する回答は次のようなものである。つまり、サブクラスで親クラスの機能を書き換えて増やすのではなく、「機能だけを別のクラスにして」それと既存のクラスを「複合させて」必要なものを組み立てる。こうすれば、M 個のクラスと N 個のクラスをそれぞれ用意すれば済むわけだ。これをコンポジションなどと呼ぶ。

では、さっきの例題を (全部作るのは大変だから途中まで) コンポジション型に手直ししてみよう。冒頭部分は変わらない。

```
import java.awt.*;
import javax.swing.*;

public class R10Sample4 extends JApplet {
  Image buf;
  boolean go;
  double time;
  Animation[] a = new Animation[20];
  int count = 0;
```

次に、図形を増やす部分はちょっと違って来ているが、どう違うかは少し後で説明する。

```
  public void init() {
    a[count++] = new FlyingMove(new Circle(Color.red, 100, 100, 15), -30, 40);
    a[count++] = new CircleMove(new Circle(Color.blue, 120, 100, 10), 30, 5);
    a[count++] = new CircleMove(new Triangle(Color.green,
                                100, 100, 140, 100, 80, 120), 40, 2);
    a[count++] = new ChgColor(new Circle(Color.red, 100, 140, 20),
                  new Color[]{Color.pink, Color.cyan, Color.black}, 0.5);
    AnimGroup g1 = new AnimGroup(160, 100);
    g1.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
    g1.add(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10));
    g1.add(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10));
    g1.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
    a[count++] = new FlyingMove(g1, 45, 35);
    AnimGroup g2 = new AnimGroup(160, 100);
    g2.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
    g2.add(new ChgColor(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10),
             new Color[]{Color.red, Color.white}, 0.7));
    g2.add(new CircleMove(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10),
             3.0, 5.0));
    g2.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
    a[count++] = new FlyingMove(g2, 25, 45);
  }
```

スレッドとか描画は前と同じ。

```
  public void update(Graphics g) {
    if(buf == null) { buf = createImage(getWidth(), getHeight()); }
    Graphics2D g2 = (Graphics2D)buf.getGraphics();
    g2.clearRect(0, 0, getWidth(), getHeight()); paint(g2);
    g.drawImage(buf, 0, 0, this);
  }
  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    for(int i = 0; i < count; ++i) a[i].draw(g2);
  }
  public void start() {
    go = true; time = 0.001 * System.currentTimeMillis();
    new Thread(new Runnable() {
      public void run() {
        while(go) {
          try { Thread.sleep(50); } catch(Exception ex) { }
          double dt = System.currentTimeMillis()*0.001 - time;
```

```
        for(int i = 0; i < count; ++i) a[i].addTime(dt);
        time += dt; repaint();
      }
    }
  }).start();
}
public void stop() { go = false; }
```

インタフェース Animation はメソッドがかなり増えている。というのは、継承を使わずに「はめ込み」で組み立てるためには、はめ込むクラスがどういうメソッドを持っているかをきちんと定義しておく必要があるから。

```
interface Animation {
  public void draw(Graphics g);
  public void addTime(double dt);
  public void moveTo(double x, double y);
  public double getX();
  public double getY();
  public void setColor(Color c);
  public Color getColor();
}
```

ではまず、円クラスを用意する。このクラスは勝手に飛んだりしない、単なる円を表している。

```
static class Circle implements Animation {
  Color cl;
  double gx, gy, rad;
  public Circle(Color c, double x, double y, double r) {
    cl = c; gx = x; gy = y; rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(cl);
    g.fillOval((int)(gx-rad),(int)(gy-rad),(int)(rad*2),(int)(rad*2));
  }
  public void addTime(double dt) { }
  public void moveTo(double x, double y) { gx = x; gy = y; }
  public double getX() { return gx; }
  public double getY() { return gy; }
  public void setColor(Color c) { cl = c; }
  public Color getColor() { return cl; }
}
```

次は「飛ぶ」機能だけをクラスとして用意する。このクラスは Animation を実装したオブジェクト (つまり図形) を 1 つ受け取り、その図形を「飛ばす」部分だけを受け持つ。それには、時間とともに gx、gy を変化させ、その位置に保持している図形を動かせばよい。これを利用する側では、FlyingMove オブジェクトを作る時に、飛ばしたい図形をパラメタとして渡してやればよい。

```
static class FlyingMove implements Animation {
  Animation anim;
  double gx, gy, vx, vy;
  public FlyingMove(Animation a, double vx1, double vy1) {
    anim = a; gx = a.getX(); gy = a.getY(); vx = vx1; vy = vy1;
  }
  public void draw(Graphics g) { anim.draw(g); }
  public void addTime(double dt) {
    anim.addTime(dt); gx += vx * dt; gy += vy * dt;
    if (gx<0.0 && vx < 0.0) { vx = -vx;}
    if (gx>400.0 && vx > 0.0) { vx = -vx;}
    if (gy<0.0 && vy < 0.0) { vy = -vy;}
    if (gy>300.0 && vy > 0.0) { vy = -vy;}
    anim.moveTo(gx, gy);
  }
  public void moveTo(double x, double y) { gx = x; gy = y; }
  public double getX() { return gx; }
  public double getY() { return gy; }
  public void setColor(Color c) { anim.setColor(c); }
  public Color getColor() { return anim.getColor(); }
}
```

こうしておけば、飛ぶ代りに円周上を動くようにするという別の機能を用意してそれと Circle を組み合わせるのも問題ない。

```
  static class CircleMove implements Animation {
    Animation anim;
    double gx, gy, rad, vtheta, theta = 0;
    public CircleMove(Animation a, double r, double vt) {
      anim = a; gx = a.getX(); gy = a.getY(); rad = r; vtheta = vt;
    }
    public void draw(Graphics g) {
      anim.moveTo(gx+rad*Math.cos(theta), gy+rad*Math.sin(theta));
      anim.draw(g);
    }
    public void addTime(double dt) { anim.addTime(dt); theta += vtheta * dt; }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { anim.setColor(c); }
    public Color getColor() { return anim.getColor(); }
  }
```

図形の方も円だけでなく三角形を増やそう。三角形も円と同様に飛んだり円周軌道で動いたりさせられる (つまり任意に組み合わせられる)。

```
  static class Triangle implements Animation {
    Color cl;
    double gx, gy, dx0, dy0, dx1, dy1, dx2, dy2;
    public Triangle(Color c, double x0, double y0, double x1, double y1,
                    double x2, double y2) {
      cl = c; gx = (x0+x1+x2)/3; gy = (y0+y1+y2)/3;
      dx0 = x0-gx; dx1 = x1-gx; dx2 = x2-gx;
      dy0 = y0-gy; dy1 = y1-gy; dy2 = y2-gy;
    }
    public void draw(Graphics g) {
      int[] x = new int[]{(int)(gx+dx0),(int)(gx+dx1),(int)(gx+dx2)};
      int[] y = new int[]{(int)(gy+dy0),(int)(gy+dy1),(int)(gy+dy2)};
      g.setColor(cl); g.fillPolygon(x, y, 3);
    }
    public void addTime(double dt) { }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { cl = c; }
    public Color getColor() { return cl; }
  }
```

もうちょっと別の機能として、「色を変化させる」ことだけをクラスとして分離したものも作ってみた。これはコンストラクタで色の配列を受け取り、その色の順番に色が変化していく。

```
  static class ChgColor implements Animation {
    Animation anim;
    Color[] colors;
    double time = 0, period;
    public ChgColor(Animation a, Color[] c, double p) {
      anim = a; colors = c; period = p;
      if(c.length == 0) throw new RuntimeException("empty color array");
    }
    public void draw(Graphics g) { anim.draw(g); }
    public void addTime(double dt) {
      anim.addTime(dt); time += dt;
      anim.setColor(colors[(int)(time/period) % colors.length]);
    }
    public void moveTo(double x, double y) { anim.moveTo(x, y); }
    public double getX() { return anim.getX(); }
    public double getY() { return anim.getY(); }
    public void setColor(Color c) { }
    public Color getColor() { return anim.getColor(); }
  }
```

さて、顔とかはどうすればいいのか? それには、「複数の図形をくっつけた図形」が作れるようにすればいい。draw() のところで不思議なことをしているが、これは部品の位置は「グループの中心位置原点とする相対座標」を入れてあるのに対し、描く瞬間だけは画面座標にしておかないと正しく描けないので、一時的に画面座標を設定し、描き終わったら元に戻すというワザを駆使しているわけである。

```
  static class AnimGroup implements Animation {
    Animation[] a = new Animation[20];
    int count = 0;
    Color cl = Color.black;
    double gx, gy;
    public AnimGroup(double x, double y) { gx = x; gy = y; }
    public void add(Animation anim) {
      if(count+1 < a.length) a[count++] = anim;
    }
    public void draw(Graphics g) {
      for(int i = 0; i < count; ++i) {
        double x = a[i].getX(), y = a[i].getY();
        a[i].moveTo(x+gx, y+gy); a[i].draw(g); a[i].moveTo(x, y);
      }
    }
    public void addTime(double dt) {
      for(int i = 0; i < count; ++i) a[i].addTime(dt);
    }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { cl = c; }
    public Color getColor() { return cl; }
  }
}
```

複合図形クラスを使って円と 3 つの三角形を組み合わせれば顔ができる。しかし動かない三角形の代わりに、色が変化する三角形とか、円周軌道で動く三角形とかを使ってもよい。このように、コンポジションを基本にしておけば継承よりもずっと用意に機能の任意の組み合わせが活用できる。今度はクラス図を描いてみても継承関係がないのが分かる (図 2)。

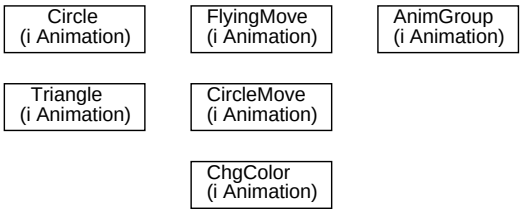

図 2: コンポジションの例のクラス図

演習 6 上のプログラムもコピーできるように用意したので、コピーしてきてそのまま動かせ。動いたら機能や図形の組み合わせを変えて別の動きをする絵を作ってみよ。もし余裕があれば、新しい機能を追加するとなおよい。

5.3 再び継承によるくくり出し+抽象クラス

コンポジションの利点は分かったが、どうも同じメソッド等を繰り返し書くのでプログラムが長くて冗長だ、と思った人もいるかと思う。そう、そこを何とかしたいですね? それには…その「共通部分」をくくり出すために継承を使えばよい。この場合は、差分プログラミングのように延々と長い継承の連鎖ができるというより、複数の類似したクラスの共通部分を 1 つの親クラスにくくり出す、という感じになる。

ところで、そのようなくくり出しを行う時、くくり出した親クラスは「共通部分の置き場所」であり、実際にインスタンスを生成することはない場合が多い。たとえば複数の図形の共通部分をくくり出した場合、その共通部分を集めた親クラスはメソッド `draw()` が定義できない (だって特定の形は持っていないから)。このようなメソッドは、インタフェースでのメソッド定義と同様、本体 (コード部分) を「`;`」だけにして、さらにキーワード `abstract` をそのメソッドおよびクラス定義の冒頭の両方に指定する。また、インタフェースで定義されているメソッドに本体 (コード) をつけない場合もこれと同じ扱いになる。[4]このようなメソッドを「抽象メソッド」(abstract method)、抽象メソッドを持つようなク

[4]時々「(クラス名) は abstract として宣言する必要があります。(メソッド指定) を (クラス名) で定義していません。」というコンパイルエラーで悩んでいる人がいるが、これはまさに前記の条件に関係するもので、インタフェースを implements したのに (スペルミスなどで) そのインタフェースが定義しているメソッドを定義し損なっている場合に出る。

ラスを「抽象クラス」と呼ぶ (分かりにくいかも知れないが、要するにインタフェースでのメソッド定義は全て自動的に abstract がつけられて処理されると思ってください)。
先の例題を上のような考え方で整理してみる。各種クラスのはじまりまでずっと同じ。

```
import java.awt.*;
import javax.swing.*;

public class R10Sample5 extends JApplet {
  Image buf;
  boolean go;
  double time;
  Animation[] a = new Animation[20];
  int count = 0;

  public void init() {
    a[count++] = new FlyingMove(new Circle(Color.red, 100, 100, 15), -30, 40);
    a[count++] = new CircleMove(new Circle(Color.blue, 120, 100, 10), 30, 5);
    a[count++] = new CircleMove(new Triangle(Color.green,
                                100, 100, 140, 100, 80, 120), 40, 2);
    a[count++] = new ChgColor(new Circle(Color.red, 100, 140, 20),
                  new Color[]{Color.pink, Color.cyan, Color.black}, 0.5);
    AnimGroup g1 = new AnimGroup(160, 100);
    g1.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
    g1.add(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10));
    g1.add(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10));
    g1.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
    a[count++] = new FlyingMove(g1, 45, 35);
    AnimGroup g2 = new AnimGroup(160, 100);
    g2.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
    g2.add(new ChgColor(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10),
             new Color[]{Color.red, Color.white}, 0.7));
    g2.add(new CircleMove(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10),
             3.0, 5.0));
    g2.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
    a[count++] = new FlyingMove(g2, 25, 45);
  }
  public void update(Graphics g) {
    if(buf == null) { buf = createImage(getWidth(), getHeight()); }
    Graphics2D g2 = (Graphics2D)buf.getGraphics();
    g2.clearRect(0, 0, getWidth(), getHeight()); paint(g2);
    g.drawImage(buf, 0, 0, this);
  }
  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    for(int i = 0; i < count; ++i) a[i].draw(g2);
  }
  public void start() {
    go = true; time = 0.001 * System.currentTimeMillis();
    new Thread(new Runnable() {
      public void run() {
        while(go) {
          try { Thread.sleep(50); } catch(Exception ex) { }
          double dt = System.currentTimeMillis()*0.001 - time;
          for(int i = 0; i < count; ++i) a[i].addTime(dt);
          time += dt; repaint();
        }
      }
    }).start();
  }
  public void stop() { go = false; }
  interface Animation {
    public void draw(Graphics g);
    public void addTime(double dt);
    public void moveTo(double x, double y);
    public double getX();
    public double getY();
    public void setColor(Color c);
    public Color getColor();
```

```
}
```

さて、「図形」という抽象クラスを用意し、ここに図形の基本的な機能を集める。このクラスでは Animation インタフェースにあるメソッド draw() を定義していないので、このメソッドは抽象メソッドになる。そのため、クラスそのものも abstract とつける必要がある。

```
static abstract class Figure implements Animation {
  Color cl;
  double gx, gy;
  public Figure(Color c, double x, double y) { cl = c; gx = x; gy = y; }
  public void addTime(double dt) { }
  public void moveTo(double x, double y) { gx = x; gy = y; }
  public double getX() { return gx; }
  public double getY() { return gy; }
  public void setColor(Color c) { cl = c; }
  public Color getColor() { return cl; }
}
```

ここでクラス図を示しておこう (図 3)。

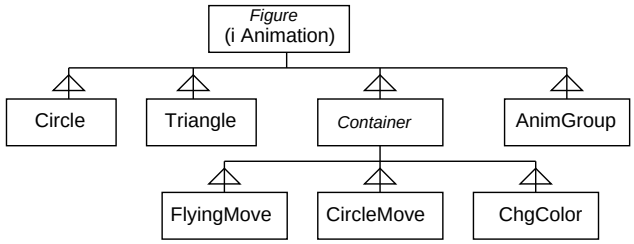

図 3: 抽象クラス+コンポジションの例のクラス図

円や三角形はこの Figure クラスから共通部分を継承して来て、独自の部分だけを定義するので簡単になる。

```
static class Circle extends Figure {
  double rad;
  public Circle(Color c, double x, double y, double r) {
    super(c, x, y); rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(cl);
    g.fillOval((int)(gx-rad),(int)(gy-rad),(int)(rad*2),(int)(rad*2));
  }
}

static class Triangle extends Figure {
  double dx0, dy0, dx1, dy1, dx2, dy2;
  public Triangle(Color c, double x0, double y0, double x1, double y1,
                  double x2, double y2) {
    super(c, (x0+x1+x2)/3, (y0+y1+y2)/3);
    dx0 = x0-gx; dx1 = x1-gx; dx2 = x2-gx;
    dy0 = y0-gy; dy1 = y1-gy; dy2 = y2-gy;
  }
  public void draw(Graphics g) {
    int[] x = new int[]{(int)(gx+dx0),(int)(gx+dx1),(int)(gx+dx2)};
    int[] y = new int[]{(int)(gy+dy0),(int)(gy+dy1),(int)(gy+dy2)};
    g.setColor(cl); g.fillPolygon(x, y, 3);
  }
}
```

同様に、1 つ図形を中に持ち、色々な動き等をつけるための抽象クラス Container を用意した。

```
static abstract class Container extends Figure {
  Animation anim;
  public Container(Animation a) {
    super(a.getColor(), a.getX(), a.getY()); anim = a;
  }
  public void draw(Graphics g) { anim.draw(g); }
  public void addTime(double dt) { anim.addTime(dt); }
```

```
    public void setColor(Color c) { anim.setColor(c); }
    public Color getColor() { return anim.getColor(); }
  }
```

Container を土台にして、違う部分だけを書くことで飛ぶ動き、円周軌道の動き、色の変化とも短く書ける。

```
  static class FlyingMove extends Container {
    double vx, vy;
    public FlyingMove(Animation a, double vx1, double vy1) {
      super(a); vx = vx1; vy = vy1;
    }
    public void addTime(double dt) {
      super.addTime(dt); gx += vx * dt; gy += vy * dt;
      if (gx<0.0 && vx < 0.0) { vx = -vx;}
      if (gx>400.0 && vx > 0.0) { vx = -vx;}
      if (gy<0.0 && vy < 0.0) { vy = -vy;}
      if (gy>300.0 && vy > 0.0) { vy = -vy;}
      anim.moveTo(gx, gy);
    }
  }

  static class CircleMove extends Container {
    double rad, vtheta, theta = 0;
    public CircleMove(Animation a, double r, double vt) {
      super(a); rad = r; vtheta = vt;
    }
    public void draw(Graphics g) {
      anim.moveTo(gx+rad*Math.cos(theta), gy+rad*Math.sin(theta));
      anim.draw(g);
    }
    public void addTime(double dt) { super.addTime(dt); theta += vtheta * dt; }
  }

  static class ChgColor extends Container {
    Color[] colors;
    double time = 0, period;
    public ChgColor(Animation a, Color[] c, double p) {
      super(a); colors = c; period = p;
      if(c.length == 0) throw new RuntimeException("empty color array");
    }
    public void addTime(double dt) {
      super.addTime(dt); time += dt;
      anim.setColor(colors[(int)(time/period) % colors.length]);
    }
    public void moveTo(double x, double y) { anim.moveTo(x, y); }
    public double getX() { return anim.getX(); }
    public double getY() { return anim.getY(); }
    public void setColor(Color c) { }
  }
```

複合図形については中に沢山の図形が入るので、Container ではなく Figure が親クラスとなっている。

```
  static class AnimGroup extends Figure {
    Animation[] a = new Animation[20];
    int count = 0;
    public AnimGroup(double x, double y) { super(Color.black, x, y); }
    public void add(Animation anim) {
      if(count+1 < a.length) a[count++] = anim;
    }
    public void draw(Graphics g) {
      for(int i = 0; i < count; ++i) {
        double x = a[i].getX(), y = a[i].getY();
        a[i].moveTo(x+gx, y+gy); a[i].draw(g); a[i].moveTo(x, y);
      }
    }
    public void addTime(double dt) {
      for(int i = 0; i < count; ++i) a[i].addTime(dt);
    }
  }
}
```

このように、継承とコンポジションをうまく組み合わせて、冗長性が小さく、組み合わせて使いやすいクラス群を作れると「美しい」と思うがいかがだろうか? ここから先は皆様も色々悩んでみていただきたい。

演習 6' 上のプログラムもコピーできるように用意したので、コピーしてきてそのまま動かせ。動いたら機能や図形の組み合わせを変えて別の動きをする絵を作ってみよ。もし余裕があれば、新しい機能を追加するとなおよい。演習 6 と演習 6' とどちらがやりやすいか考えてみよう。

A　本日の課題 10A

「演習 2〜4」のうち好きなものについて作成したプログラム 1 つのソースコードをいつも通り、「本日中に」久野までメールで送付してください。題材がアプレットではないので、ソースコードのみで結構です。具体的な内容は次の通り。

1. Subject: は「Report 10A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 選んだプログラム 1 つのソース。
4. その簡単な説明。
5. 下記のアンケートの回答。

   Q1. MIDI データ (楽器+音程+タイミング) の原理は納得しましたか?
   Q2. 「抽象データ型」という考え方についてはどうですか?
   Q3. その他感想、冬休みレポートの構想などについてひとこと。

B　次回までの課題

ありません。冬休み課題をよろしく。