オブジェクト指向プログラミング'10 ＃1

久野　靖*

2010.9.6

本科目「オブジェクト指向プログラミング」では、今日のプログラミングにおける主要なパラダイムの１つであるオブジェクト指向が、具体的にどのようにプログラミング言語に取り入れられ、それによってこれまでになかったどのようなプログラミングスタイルが可能になっているのかを具体的に見ていきます。

言語としては Java を使用し、実際にさまざまなプログラムを Java で書いていただきながら、上記の点を確認して行きたいと思います。評価は、毎回の出席+次回までの課題と最終レポートによるものとします。重要なことですが、ある程度時間を使ってプログラムを書いてみないと、この科目の内容は身につきません。このため、授業時間に加えて、週末に 2〜3 時間くらいは課題をやるために使用してください。

毎回のテーマは、おおよそ次のように考えています。細かいところは、進捗によって変わってくるかと思います。

(1) オブジェクト指向の概念、Java 言語の入門、既存クラスの利用、グラフィクスオブジェクト、Java 言語の制御構造

(2) クラスを定義する、オブジェクト指向グラフィクス、入力イベントの扱い、多相性の活用

(3) スレッドとアニメーション、時間経過のロジックと画面変化、クラス群の組み合わせによるドメイン固有言語

(4) GUI 部品、ユーザインタフェースの設計、CUI、ファイル入出力

(5) 実用プログラムの要素、レイアウトマネージャ、メニュー、総合制作

最終課題は、それまでに学んだことがらを活用して好きなプログラムを制作してもらうことにしますので、毎回の演習をちゃんとやっていただければ、苦労せずに (むしろ楽しく) こなせると思います。では 5 週間という短い間ですが、よろしくお願いします。

1　オブジェクト指向プログラミング

プログラミングをするとき、プログラムを書き下す「書き方」のことを**プログラミング言語**と言います。プログラミング言語には、さまざまな流派があります。この本でこれから学ぶのは、**オブジェクト指向**と呼ばれる流れを汲む、Java という名前の言語です。先に一言でイメージしてもらうとする

*経営システム科学専攻

と、オブジェクト指向というのは「もの (オブジェクト)」を中心とした「考え方」のスタイル、とでも言っておきましょうか。これが何でありどう嬉しいか、について説明しておきます。

手続き型とオブジェクト指向

オブジェクト指向流プログラミングが出現する以前のプログラミングは手順 (手続き) を中心とした考え方でしたから、

> これをするには、あれをして、これをして、それをする。あれをするには、ああして、こうして、こうする。

というのを積み上げてプログラムを作っていました (図 1)。つまり逐一順番に、コンピュータが処理することを書き下し、命令してコンピュータを動かしていたのです。コンピュータはプログラマが書いた命令に従って動くべきものですから、これは自然なことではあります。しかしこの方法は、複雑な問題をうまくプログラムする上で、しだいに限界に近づいて来たのです。

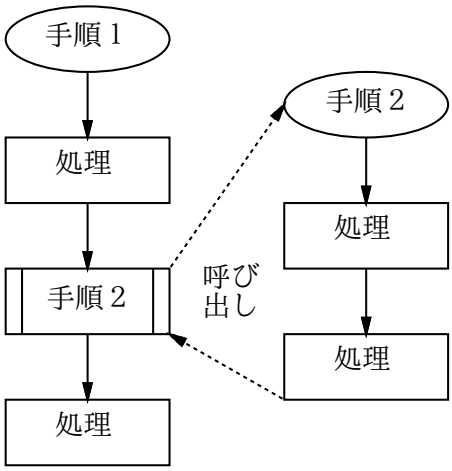


図 1: 手順に基づくプログラミング

そこで登場したのがオブジェクト指向プログラミングです。オブジェクト指向では「もの」が中心となりますから、

> このプログラムでは、ああいうものと、こういうものと、そういうものがある。ああいうものは、こういう動作と、こういう動作ができる。

という形でプログラムを書くことになります (図 2)。そんな書き方で、コンピュータが具体的にどんな順番で何を処理するのか、不思議ですか? これから、その疑問について、明らかにして行きたいと思います。

以下の説明は、(言語ごとに書き方などは異なりますが) オブジェクト指向プログラミング言語に共通する原理についてまず理解していただき、オブジェクト指向のプログラムが動く感じをつかんでいただくためのものです。ですから、内容が抽象的に感じられ、ぼんやりとしか分からないかもしれませんが、次節以降で具体的に Java のプログラムを題材として確かめられます。本節はざっと読んで先に進み、その後も必要な都度、ここの説明に戻って来るようにしてみてください。

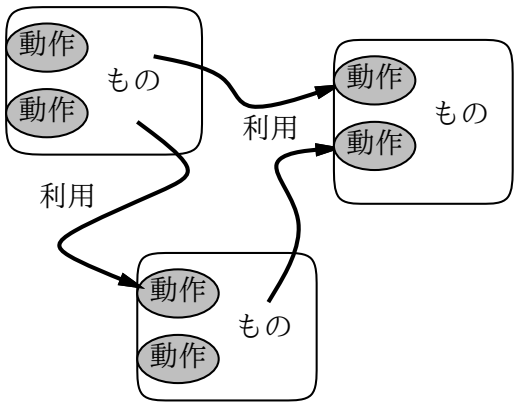


図 2: オブジェクト指向プログラミング

オブジェクト指向プログラミングの利点

では、オブジェクト指向によるプログラミングが、どのような点で優れているかについて、大雑把にまとめておきましょう。

> このプログラムでは、ああいうものと、こういうものと、そういうものがある。ああいうものは、こういう動作と、こういう動作ができる。

ということをプログラムとして書いていけばよいのですから、すべてのプログラムを 1 から作らなくても、前もっていろいろな種類の「もの」を用意しておき、それらの「もの」を組み合わせてその動作を呼び出すだけで、かなり高度なプログラムが作れるようになります。[1)]

1)用意されていない「もの」は自分で用意するので、そのときの「こういう動作」の中身はやはり、ある程度「手順」的に作りますが、その中も「用意されているもの」を利用することで、楽に作れるのです。

言い替えると、オブジェクト指向言語では、プログラムを「もの」という「かたまり」に分けて考えることで、プログラムを扱いやすくし、また、既に作られているものを利用しやすくしているのです。

もう 1 つのオブジェクト指向の嬉しいことは、かたまりの単位が「もの」なので、それを私達の日常になぞらえて考えることで「考えやすさ」「分かりやすさ」が増すという点があります。プログラムは少し高度になると「複雑」で「考えにくい」ものになりがちですから、その「考えやすさ」や「分かりやすさ」を増す、というのは重要です。

2 オブジェクト指向プログラミング言語の基本概念

具体的な Java のプログラムを見る前に、オブジェクト指向プログラミングで必須[2)] の概念である「クラスとインスタンス」についての説明からはじめて、「メソッド」や「変数」、効率的なクラス定義に欠くことのできない「継承」や「インタフェース」など、オブジェクト指向プログラミング言語の基本概念について一通り説明して行きます。[3)]

2)実は、クラスを持たない**プロトタイプ方式**と呼ばれる種類のオブジェクト指向言語もあります。クラスを持つ方を学んでおけば、プロトタイプ方式もすぐ使えるでしょう。

3)言語によって細部が違うこともありますが、そのような場合は Java ではどうなっているかに基づいて説明します。

クラスとインスタンス

Java をはじめ、多くのオブジェクト指向プログラミング言語では、プログラムを作るということは、最終的に「自分が必要とするような動作を持っ

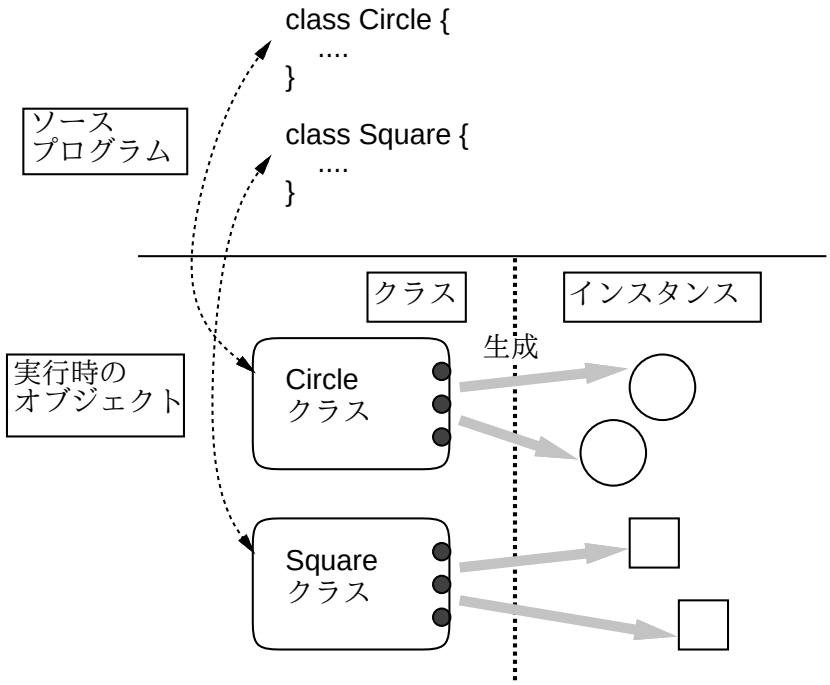


図 3: クラスとインスタンス

た「もの (**オブジェクト**)」を作り出す」ことになります。具体的にプログラムで「もの」を作り出すには、次の順で記述します。

(1) こんな種類の「もの」があるよ

(2) 実際にどの種類の「もの達」がどんな風に存在するよ

こうして、自分が解きたい問題を表現する世界について書き表していくのが、オブジェクト指向のプログラムです。

(1) の「もの」の種類についての記述は、**ソースプログラム** (プログラムを書き下したもの) の上では、**クラス** と呼ばれる単位で行ないます。[4] ソースプログラム中にある 1 つひとつの「クラス」が、実行時に利用可能になります。そして、実行時に (2) で作り出す「もの達」は、それぞれが 1 つのクラスを「ひな型」として作り出されます。これをクラスと対比して**インスタンス**と呼びます。とすると、オブジェクトはクラス (に対応するオブジェクト) である場合と、どれかのクラスからつくり出されたインスタンスである場合とがあるわけです (図 3)。

「もの」を作る時、つまりクラスを定義する時には、何を書き下せばよいのでしょう? それは次の 2 つです。

(1) どんな情報を持っているか

(2) どんな動作を持っているか

上記の 2 点を明らかにしつつ、「もの」を記述していくのが、オブジェクト指向言語によるプログラミングのスタイルです。

「もの」は一般に複数あって、ある「もの」の下請けの部品となる「もの」も一緒に作らなければならなかったり、ある「もの」を使う上位部品の「もの」を用意しなければならなくなったりします。そのような、プログラム中

4)このような、クラスに基づいてオブジェクトを定義する方式の言語を「**クラス方式**のオブジェクト指向言語」と呼びます。このほかに、「お手本」となるオブジェクトが予め用意されていて、それをコピーすることで個々のオブジェクトを作り出す方式もあり、こちらは「**プロトタイプ方式**のオブジェクト指向言語」と呼ばれます。

のオブジェクト群の設計作業が、後でプログラムの出来不出来に関係してくる、プログラミングの重要な一工程となります。

クラス定義

抽象的な説明では分かりにくいので、例題を考えてみましょう。「画面に円や正方形などさまざまな形のものが表示される」プログラムを考えます。すると、だとえば「円」を「もの」として扱いたいので「円」を表すクラスを作ろう、ということになります (正確な書き方はまた後で出て来ますので、ここでは雰囲気だけつかんでください)。

```
class Circle {
  ...
}
```

クラスを作る時には、そのクラスで覚えておくべき「情報」と、それらの情報に基づいて行なう「動作」の両方を決めなければなりません。それが「...」のところに書く内容になります。

円クラスを作るときには、任意の円を画面に表示できるようにするために必要な情報と動作を、円クラスの持ち物として定義します。「情報」の方は、「変数」という形で記憶します。[5]「動作」の方は、「メソッド」という形で定義します。つまり次のようになるわけです。

```
class Circle {
  変数の定義…
  メソッドの定義…
}
```

5)プログラミングにおける「変数」というのは、数学の変数とは少し違っていて、「値を入れておける入れもの」であると考える方がぴったりします。

インスタンス変数

円クラスの定義を鋳型のように使って、画面に表示される 1 つひとつの具体的な円インスタンスが生成されます。とすれば、鋳型である円クラスには、表示する円についてのどのような定義を書けばよいでしょうか。1 つの円ごとに必要な「情報」を考えましょう。円は幾何学的には、その中心座標 (x, y) と、半径 r で表されるでしょう。また、「色」もそれぞれの円に固有の情報です。

これらの「情報」を覚えるのに、プログラミング言語では**変数**というものを用います。[6] Java では、変数を定義するときは必ず、変数の**型** (値の種別を表すもの) を明示します。[7]また、**変数名**は 1 文字とは限らず、分かりやすいように名前をつけて構いません。次の例を見てください。[8]

```
class Circle { // 円を表すクラス
  int xpos, ypos, rad; // 座標と半径
  Color col; // 円の色
  ...
}
```

xpos、ypos、rad は「整数 (integer)」の値を格納する変数、col は色を格納する変数として定義されています。このように、それぞれのインスタンスに固有の情報を格納する変数を**インスタンス変数**と呼びます (図 4 右)。

6)プログラミング言語でいう変数は「さまざまなものが入る」という点で数学でいう変数と似ていますが、プログラムの実行につれて値を変更して行くという点ではやや異なります。

7)このような言語を**強い型**の言語、と呼びます。Java も強い型の言語の 1 つです。

8)「//」から右側は**コメント** (注記) といい、プログラムに関する説明を記入したものです。実際に Java でプログラムを作るときも活用します。ちょっと込み入った処理は「何をやってるか」コメントでメモっておくと後で分からなくなったりしません。

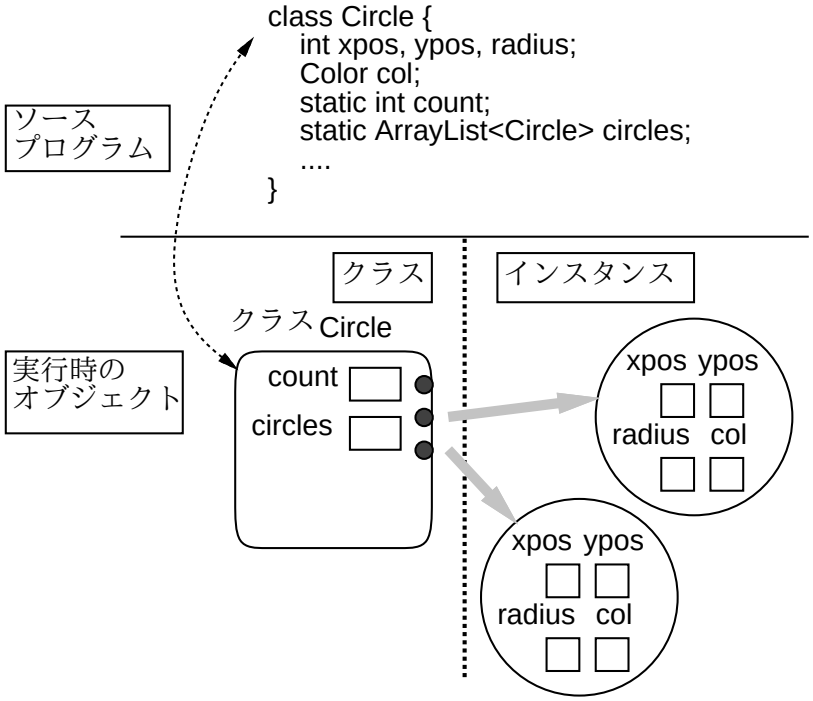


図 4: インスタンス変数とクラス変数

クラス変数

オブジェクトが持つ「情報」についてもう少し考えましょう。「情報」は、インスタンスごとに持つのが適しているものばかりではありません。たとえば、これまでにいくつの円を作ったかとか、すべての円をまとめて黒くする、という動作が必要だとします。すると、今ある円の個数はいくつかとか、それらの円は具体的にどれとどれか、という情報が必要です。このような情報は、クラス `Circle` 全体として 1 つだけあれば済みます。このような変数は**クラス変数**と呼ばれ、`static` というキーワードをつけて表します (図 4 左)。[9]

9) `ArrayList<Circle>` というのは、`Circle` を複数保持しておけるような種類のオブジェクトです。このような `<...>` のついたクラスについては、後で取り上げます。あと、実は `ArrayList<Circles>` は「いくつ円が入っているか」を記録しているので、個数を別の変数に覚えておく必要はないのですが、ここでは説明の分かりやすさのために変数 `count` も別にあることにしました。

```
class Circle {          // 円を表すクラス
  int xpos, ypos, rad; // 座標と半径
  Color col;           // 円の色
  static int count;    // 円の個数
  static ArrayList<Circle> circles; // 円を全部覚えておく変数
  ...
}
```

インスタンスメソッド

メソッドの説明に移りましょう。画面に図形を描画するプログラムを考えているのですから、まずそれぞれの円は (自分を) 画面に描く、という動作を持ちます。そのためにメソッド `draw()` を用意するとして、Java の書き方では次のようになります。[10] `draw()` の動作定義のところでは、画面に絵を描くための別のクラス定義を利用して、自分 (円インスタンス) が持っている

10) 本当の詳細な書き方は今は気にしないで、雰囲気だけつかんでください。

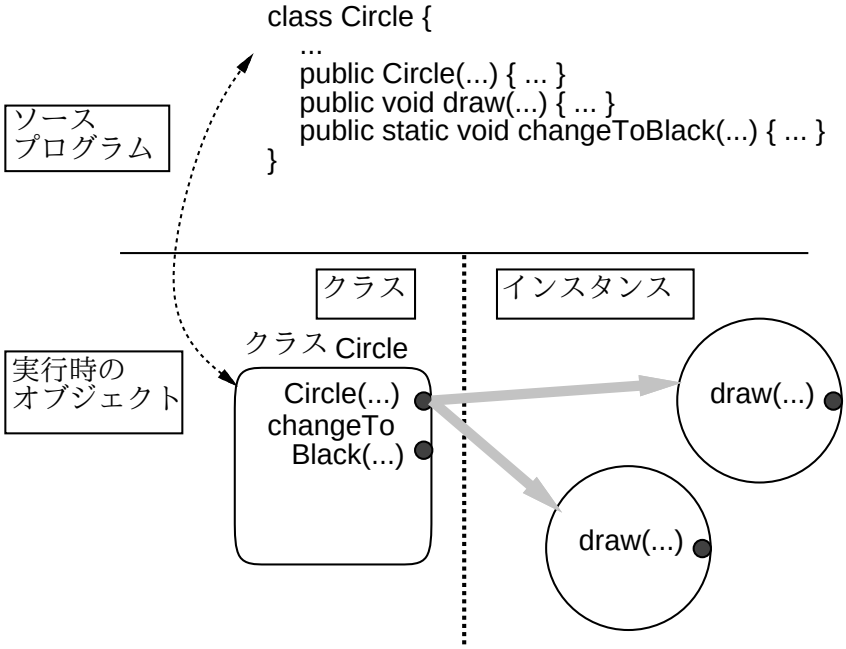


図 5: インスタンス/クラスメソッド、コンストラクタ

中心座標と半径、色の情報に基づいて円を画面に描く手順が定義されます。このような、個々のインスタンスが持つメソッドを**インスタンスメソッド**と呼びます (図 5 右)。

```
class Circle {          // 円を表すクラス
  int xpos, ypos, rad; // 座標と半径
  Color col;           // 円の色
  static int count;    // 円の個数
  static ArrayList<Circle> circles; // 円を全部覚えておく変数
  ...
  public void draw(…) { 動作… }
  ...
}
```

メソッドの定義が `public void` で始まっていますね。**public** というのは、このメソッドがクラスの外から呼べる (**公開メソッド**である)、という意味です。**void** というのは「ない」という意味の英語で、Java では「このメソッドは値を返さない」という意味で使っています。`draw()` の仕事は画面に描くことですから、特に値を返す必要はないわけです。

クラスメソッド

今定義しようとしている円クラスでは、全部の円を黒くするという動作も必要でしたね。これをメソッド `changeToBlack()` として用意します。ただし、全部の円を黒くするメソッドは、1 つひとつの円インスタンスにあるのではなく、クラス `Circle` 中に 1 箇所だけに存在する方が正しそうです。このような、クラスが直接持つメソッドを**クラスメソッド**と呼びます。クラスメソッドの定義には、変数と同様 `static` というキーワードをつけます。

```
class Circle {          // 円を表すクラス
  int xpos, ypos, rad; // 座標と半径
  Color col;           // 円の色
  static int count;    // 円の個数
  static ArrayList<Circle> circles; // 円を全部覚えておく変数

  public void draw(…) { 動作… }
  public static void changeToBlack(…) { 動作… }
  ...
}
```

メソッド呼び出し

メソッドの動作定義の中身はまだここでは考えませんが、オブジェクトのメソッドを呼び出して動作を実行させることについては考えておきましょう。インスタンスメソッドである `draw()` は、「この円を」描くのですから、次のように必ずオブジェクトを指定して呼び出します。

```
c1.draw(…);
c2.draw(…);
```

ただし、`c1` や `c2` という変数に `Circle` クラスのインスタンスが格納されているものとします。[11]これで、円 `c1` や円 `c2` が画面に描かれます。このように、個々のインスタンスを指定して呼び出すメソッドなので、インスタンスメソッドと呼ぶわけです。

これに対し、「全部を黒くする」というのは、特定の円ではなく円すべてに対してのものなので、次のようにクラス名を指定して呼び出します。

```
Circle.changeToBlack(…);
```

これで、クラス変数 `circles` に入れてあるすべての円が黒に変ります。このような、クラスに対して呼び出すメソッドのことを、クラスメソッドと呼ぶわけです。

ところで、上の例でかっこの内側の「…」は何でしょう? それは、たとえば円を描くとすれば、画面上の窓の中に描いたり、紙の上に描いたり、[12]色々な場合があります。ですから、この円を「どこに描く」という情報を渡すわけです。つまり「…」の部分にはメソッドの動作を調整するために渡す情報を指定します。これを**パラメタ** (ないし**引数**) と呼びます。メソッドによっては、パラメタが「ない」場合もあります。[13]

11) どうやってそうするかは、すぐ後で述べます。

12) つまりプリントアウトする時です。

13) たとえば、全部の円を黒くするのには、余分な情報を指定する必要はなさそうですから、実際には「`changeToBlack(…)`」の「…」の部分はなくて、「`Circle.changeToBlack()`」となるでしょう。

インスタンスの生成

これで、任意の大きさの円を画面の任意の位置に任意の色で描画することができるようになるでしょう。でも、一番大きな謎が残っています。さまざまな円のインスタンス `c1`、`c2`、…はどうやって作ったのでしょうか? それには、典型的には次のようにします。

```
Circle c1 = new Circle(Color.RED, 100, 50, 30);
Circle c2 = new Circle(Color.BLUE, 120, 90, 50);
```

これらの行は「=」の左側で変数を定義し、「=」の右側でその初期値を計算し、それを定義した変数に格納します。[14] ですから上の疑問に答えるには、「=」の右側に注目してください。ここでは、Java の **new 演算子**を使って新しいオブジェクトを作っています。つまり、「new クラス名 (…)」という式によって、指定したクラス (ここでは Circle) のインスタンスが、新しく作り出されるのです。[15]

14)「=」の意味については、2.3 節でもっと詳しく説明します。

15)円のインスタンスを作って使うという記述は、Circle クラスの**外**、つまり Circle クラスを利用するクラスのどこかに書かれている、という点を覚えておいてください。

コンストラクタ

「new Circle(…)」の「…」にはその円を初期設定するためのパラメタが書かれていますが、これを受け取って実際に初期設定をする動作もクラス中になければなりません。この、new でインスタンスを作り出す時の初期設定は、**コンストラクタ**という特別なメソッドで行ないます。コンストラクタはクラスと同じ名前 (ここでは Circle) を持っているので、それと分かります (図 5 左)。

```
class Circle { // 円を表すクラス
  int xpos, ypos, rad; // 座標と半径
  Color col; // 円の色
  ...
  public Circle(Color c, int x, int y, int r) {
    col = c; xpos = x; ypos = y; rad = r;
    ...
  }
  ...
}
```

このように、コンストラクタの典型的な仕事の 1 つは、渡されたパラメタに基づいてインスタンス変数を初期設定することです。

コンストラクタを複数書くことで、あるクラスのインスタンスを作り出す複数の方法を用意できます。たとえば円クラスの場合、中心の XY 座標と半径を指定して作り出す代りに、2 点の XY 座標を指定して、その 2 点を結ぶ線分を直径とする円を作る、という指定も考えられますね。[16]

16)2 つ以上のコンストラクタがある場合、それらは呼び出し方で区別できなければなりません。Circle の例では、2 つのコンストラクタはパラメタの数が違いますから区別できます。

```
class Circle {          // 円を表すクラス
  int xpos, ypos, rad; // 座標と半径
  Color col;           // 円の色
  ...
  public Circle(Color c, int int, doule y, int r) {
    col = c; xpos = x; ypos = y; rad = r;
    ...
  }
  public Circle(Color c, doulle x1, int y1, int x2, int y2)
    // 2 点 (x1,y1)、(x2,y2) から中心と半径を計算して初期化
    ...
  }
  ...
}
```

ここまでで、クラス Circle の定義は大まかに次のような形をしていることが決まりました。

```
class Circle {          // 円を表すクラス
  int xpos, ypos, rad; // 座標と半径
  Color col;           // 円の色
  static int count;    // 円の個数
  static ArrayList<Circle> circles; // 円を全部覚えておく変数

  public Circle(…) { 動作… }
  public Circle(……) { 動作… }
  public void draw(…) { 動作… }
  public static void changeToBlack(…) { 動作… }
  ...
}
```

これと同様に、四角形や三角形のクラスを定義して、それらのインスタンスを作り出すことで画面に様々な図形を描く、というふうにしてプログラミングを進めて行くわけです。[17)]

17)実は今回は、とりあえずプログラミングを体験してもらうため、もっと簡便な方法で図形を描きます。しかし次回からは、きちんとここで説明した方法になりますのでご心配なく。

3 継承とインタフェース

オブジェクト指向でのクラスという概念が強力なのは、1 つにはクラスという単位で分割することで、私達にとって考えやすい「ものの種別」ごとに分けてプログラムを考えられるという点です。次のステップとして、(適切に分割された) クラスどうしの間に継承という関係を持たせることで、クラスという概念をさらに強力な道具にできます。また、Java では継承をより抽象化して扱えるインタフェースという機能も用意されています。これらについて説明しましょう。[18)]

18)この節の内容は、だいぶ抽象的で難しく感じられるかも知れません。この節の内容を実際に使うのは次回以降ですから、とりあえずこの節はスキップしておいて、後で改めて読んでいただく、ということでもかまいません。

継承

再び円クラスの例で考えていきましょう。クラスを作って完成した後で、それをもっと強力なものに改造したいと思うことがあります。たとえば、上で作った円をもとに「動きまわる円」や「点滅する円」を作る、ということを考えます。そのとき、クラス Circle を直接手直ししてしまう、というのはよくない方法です。もしそうすると、「円」クラスが「動き回ることができて点滅することもできるがどっちもしないかもしれない円」というふうに、どんどん複雑になってしまうからです。

そこでオブジェクト指向言語では、あるクラスを「土台にして」別のクラスを作る、という機能を提供します。これを**継承** (inheritance) と呼び、土台となるクラスを**親クラス** (**スーパクラス**)、それをもとに作るクラスを**子クラス** (**サブクラス**) と呼びます。Java では継承を行なうには、サブクラスの冒頭に **extends** という指定を入れます。

```
class MovingCircle extends Circle {
  double vx, vy;
```

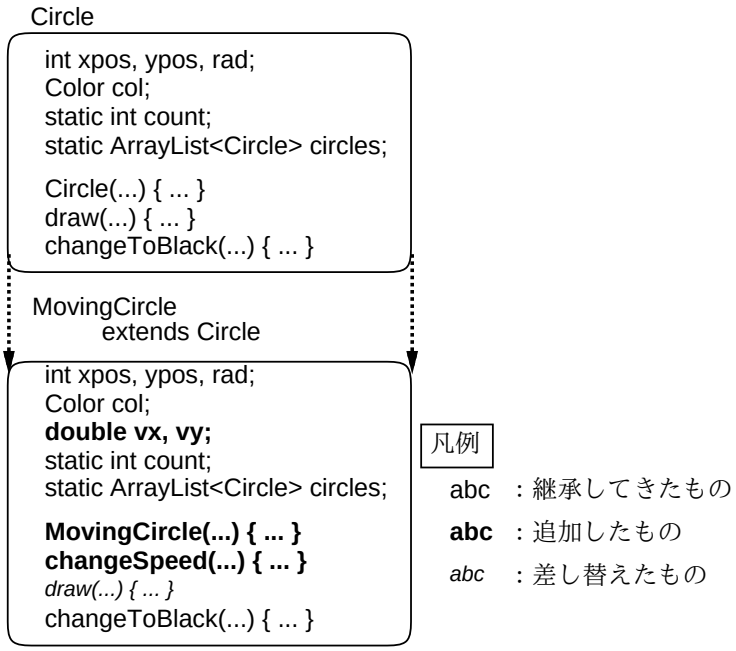


図 6: 継承

```
  public MovingCircle(…) { 動作… } // コンストラクタ
  public void changeSpeed(…) { 動作… } // 追加メソッド
  public void draw(…) { 動作… }    // 差し替えメソッド
}
```

こうすると、クラス MovingCircle はクラス Circle で定義している変数とメソッドをすべて取り込み、さらにそれに加えて自前のインスタンス変数 vx と vy、および自前のインスタンスメソッド changeSpeed() を追加したものになります。[19] また、draw() は Circle に同じ名前のものがありましたから、その場合は**差し替え** (**オーバライド**とも呼びます) になります (図 6)。なぜ差し替えたいかの説明は、すぐ出て来ます。

このように、継承を使うと、既にあるクラスはそのままで、それを土台にしてもっと機能を増やしたクラスを作ることができます。もう 1 つ興味深い機能として、MovingCircle は Circle の「1 種である」ものとして扱える点があります (図 7)。

19)コンストラクタはクラスと同じ名前なので、必ず作り直しになります。

```
Circle c3 = new MovingCircle(…);
...
c3.draw(…);
```

つまり、変数 c3 には Circle オブジェクトでも MovingCircle オブジェクトでも入れられ、どちらのオブジェクトでも「まとめて」扱えるのです。これはちょうど、「猫なら何でも預ってくれるショップ」に「ペルシャ猫」を預けるのが OK、のようなものだと思えばよいでしょう。

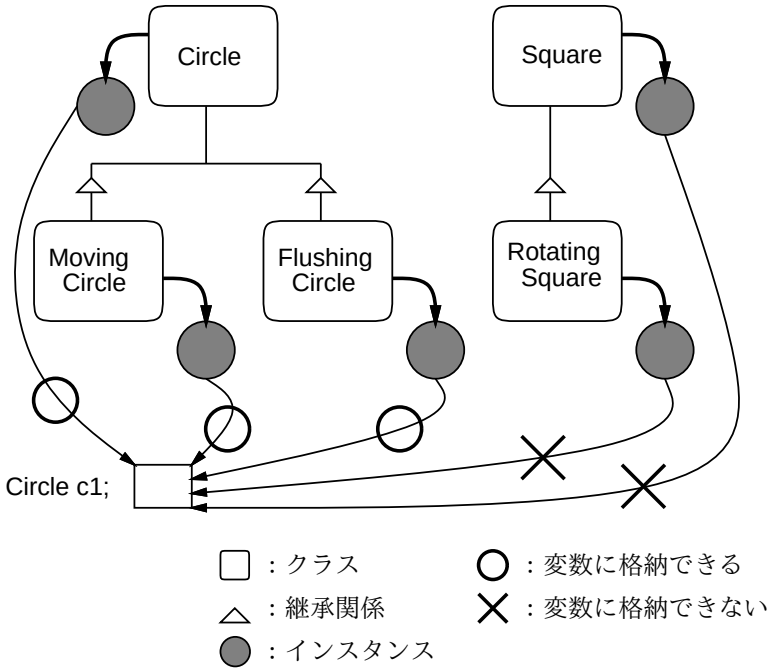


図 7: 継承関係と変数への格納

インタフェース

Java にはもうひとつ、**インタフェース**と呼ばれる重要な機能があります。これは、継承のように変数やメソッドを取り込んでは来ないけれども、上で述べたような「まとめて扱う」機能を提供するものです。たとえば、画面に色々なものを描くプログラムでは、その色々なものを形に関わらず「まとめて」扱いたいでしょう。

つまり、円か三角か四角かは気にせずに「これから表示する図形」を扱えれば、便利ですし、考えやすくなるはずです。たとえば次のような状況です。

```
Figure f1 = new Circle(…); // とりあえず f1 は円
...
if(…) { f1 = new Square(…); } // 条件によっては正方形に
...
f1.draw(…); // 実行の経過次第で円か正方形が描かれる
```

このようなことを実現する Java の仕掛けは、次の通りです。

Java のインタフェースはクラスとよく似た書き方をしますが、ただしメソッドは名前と使い方だけで動作を定義しません。また変数も初期値を与えてその後は値が変更できないものしか定義できません。[20]

[20]これを **final 変数**と呼びます。`final` はクラス変数やインスタンス変数にも指定できます。

```
interface Figure {
  public void draw(…);
}
```

ここではメソッド `draw()` だけが定義されていますが、このメソッドが `Figure` インタフェースに従うオブジェクトの共通の切り口になるのです。つまり、`Figure` インタフェースの仲間にまとめられるのは、メソッド `draw()` の実

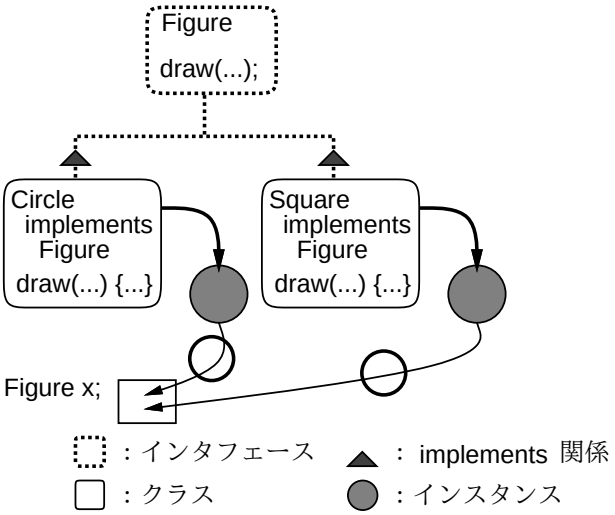


図 8: インタフェースと変数への格納

現を持つオブジェクト (すなわち、何かを描くことのできるオブジェクト)、という性質だけは共通に持たなければなりません。

`Figure` インタフェースに従うオブジェクトを作るには、これまで通りクラスを定義しますが、その冒頭で **implements** という指定により、そのクラスが `Figure` インタフェースに従うことを示します。

```
class Circle implements Figure {
  ...
}

class Square implements Figure {
  ...
}
```

これによって、`Circle` オブジェクトも `Square` オブジェクトも `Figure` インタフェース型の変数に入れて扱えるようになるわけです (図 8)。図 7 と図 8 を見比べて、変数へのオブジェクトの格納についてイメージをつかんでください。継承とインタフェースはよく似ています。[21)]

21)重要な違いは、継承では変数やメソッドを「必ず取り込んで来てしまう」のに対し、インタフェースの `implements` はそのようなことがない (こういうメソッドを持つよ、という約束だけをする)、という点です。

型の判定と行き来

ところで、複数のクラスのインスタンスをまとめて扱っている中で、「実際にはそのうちのどのクラスか」を調べたり、そのクラス固有の機能を使いたいことがあります。たとえば「ここまではどの猫も同じに扱ったけど、この猫がペルシャなら特にこうしよう」ということですね。そのために、次の 2 つの機能があります。

- インスタンスがどのクラス (以下) か調べる **instanceof 演算子**
- 型を変換する**キャスト**機能

具体的には、たとえば次のようになります。[22)]

22)`if` については後で詳しく説明します。

```
Circle c3 ...
...
if(c3 instanceof MovingCircle) { // c3の円が「動く円」なら
  ((MovingCircle)c3).changeSpeed(…);  // 速度変更
}
```

このように、「オブジェクト instanceof 型」でオブジェクトがどの型にあてはまるかを判別でき、また「(型) 式」という構文 (**キャスト**) を使うことで、これまで Circle として扱っていたものを MovingCircle としての扱いに戻せます。[23]

[23] MovingCircle でなければ changeSpeed() メソッドは使えないことに注意。なお、これらの書き方はインタフェースに対しても使えます。

4 再び手続き型プログラミングについて

ここまでで、オブジェクト指向プログラミングのさまざまな側面について解説して来ました。しかし、まだ残っていることがあります。それは、メソッドの中の「動作」は結局どうやって指定するのか、ということです。その答えは…手続き型プログラミングを使って、(場合によっては) オブジェクトではないデータを処理することで、というものです。だまされた、と思いましたか? このことを少し説明しましょう。

手続き型の持つ意味

上で述べた、「動作」は手続き型で指定する、ということは、ある意味では当然のことなのです。私達が使っているコンピュータの心臓部である **CPU**(Central Processing Unit、中央処理装置) は、次のような原理で動作しています。

- **命令**を 1 つずつ、順番に実行していく。
- 各命令は、**メモリ** (主記憶装置) との間でデータをやりとりしたり (転送命令)、コンピュータ外部との間でデータをやりとりしたり (入出力命令)、CPU 内部でデータを加工したり (演算命令)、データの値に応じて命令の実行順序を (「順番に」以外のやり方に) 変更する (制御命令)、などの機能を持つ。

この「命令の並んだもの」がプログラムですから、プログラミング言語は一番下の (細かい) レベルではこの命令に対応しなければなりません。また、命令が直接扱うようなデータはオブジェクトのような複雑な構造は持たない、「裸の」データです。このようなデータを Java では**基本型**のデータ (ないし値)、と呼びます。

たとえば、次のようなソースプログラムを見てみましょう。[24]

[24] while も後で詳しく扱います。

```
int x = 10;     // 整数を入れる変数 x を用意し、10 を格納。
int f = 1;      // 整数を入れる変数 f を用意し、1 を格納。
while(x > 0) { // x が 0 より大きい間以下を繰り返し、
  f = f * x;    // f の値と x の値を乗じたものを f に格納。
  x = x - 1;    // x の値から 1 引いたものを x に格納。
}               // ここまでを繰り返す。
// ここで f には 10 の階乗が計算されている。
```

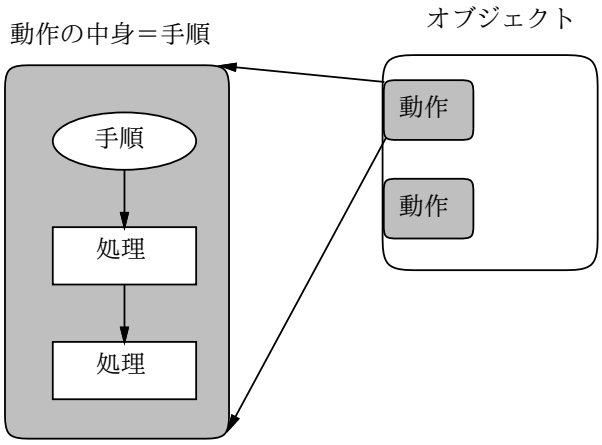


図 9: オブジェクトと手続き型記述の関係

ここで x や f はメモリのどこかの場所に対応し、「10」や「1」などは基本型の値、「=」は転送命令、「*」や「-」などは演算命令に対応し、while とそこに書かれている条件などは制御命令に対応すると考えられます。そして、これらは原則として (制御命令に対応する場合以外は) 書かれている順番に実行されていくわけです。[25)]

手続き型のプログラミングは大きな成功を収めて広く使われましたが、ソフトウェアが複雑高度になるにつれて限界が明らかになり、それに呼応してオブジェクト指向のようなより新しい考えが現われ発展してきたのです。ですから、オブジェクト指向言語でも一番下の計算レベルでは基本型のデータを主に用い、手続き型の枠組みに従っているわけです (図 9)。

25)だったらコンピュータの命令そのものを書けば良いのでは、と思われましたか? コンピュータができた最初の頃はそうしていたのですが、そのようなプログラム (機械語のプログラム、と呼びます) は大変繁雑で分かりにくいものでした。そこで、それをもっと抽象化して人間になじみやすい形にしたのが、上に示したような手続き型のコードです。

5 Java プログラミング環境の入手

本書では Java プログラムを動かすのに「JDK」(Java Development Kit) と呼ばれているツール群を使います。JDK は Oracle 社[26)]のサイトからネット経由で入手できます。次の URL の Web ページからたどって見てください。

```
http://java.sun.com/javase/
```

26)Java を開発した本家本元である Sun Microsystems 社は 2010 年に Oracle に買収されました。

本書の執筆時点での最新版は、Windows、Solaris、Linux 環境で JDK 1.6 が使えます。[27)]自分の使っている環境に合わせて JDK を選んで入手してください。

JDK を取り寄せて展開すると、javac、java、appletviewer などのコマンドが使えるようになるはずです。[28)]動くかどうか確認するためには、java -version というコマンドを実行してみてください。次のように、JDK のバージョン番号が表示されるはずです。

```
% java -version
java version "1.6.0_21"
...
%
```

27)本書の例題は 1 つ前のバージョンである JDK 1.5 でも同様に動きますから、最新版でなくても大丈夫です。でも性能などを考えると、できるだけ新しい版を利用した方がよいと思います。

28)もしかしたら、展開して取り出したコマンドが置いてある場所を、自分の「実行パス」に含めるように設定を追加する必要があります。

また、JDK にどのようなクラスライブラリが備わっているかの文書、API ドキュメントと呼ばれるものも入手する必要があります。[29]こちらもネット経由で取り寄せられます。必ず、自分が入手した JDK の版に合ったものを用意してください。よく探せば日本語で書かれたものも見つかると思います。こちらは展開すると HTML ファイル群になりますから、その入口のページを手持ちの Web ブラウザの「ファイルを開く」で表示させればよいのです。その先はブラウザ上でリンクをたどっていくだけで必要なページを探すことができます。こちらも、ちゃんとドキュメントが表示できることを確認してください。

29)これがなクラスが使え不便です。

6　はじめてのプログラムを動かす

どのプログラミング言語を学ぶときも、最初は、「何か画面に出力するプログラム」を動かしてみるのが定石です。これは入力が一切なく、[30]出力だけがあるという、やや例外的な形のプログラムです。[31]

30)正確にいえば、入力はプログラムに予め埋め込んであると考えることができます。

31)ところで、逆に出力がないプログラムというのは決してあり得ません。プログラムが実行され、何か作業を行ったら、必ず何かの結果を画面なりどこか別の場所なりに出力しなければいけません。出力がなければ、ユーザ (そのプログラムを動かしてみた人) は何も得るものがないからです。

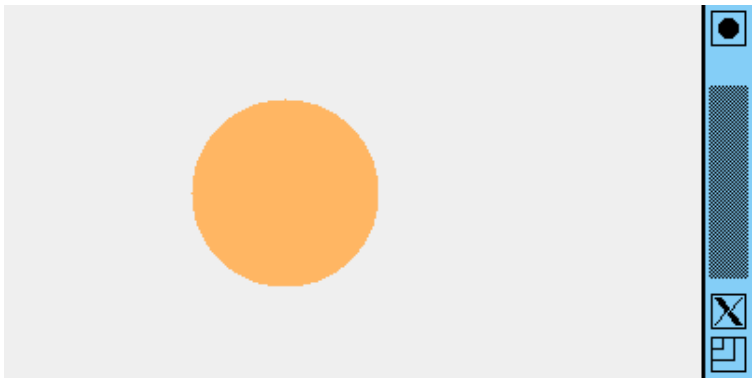


図 10: 円を描くプログラムの画面

記念すべき皆様のはじめての Java プログラムは、画面に窓が出て、その中に円が描かれている、というものです。動かしているところを、図 10 に示しました。さっそく、プログラムを見てみましょう。

例題 2-1: 画面に円を描く

```
import java.awt.*;    // 1
import javax.swing.*; // 1

public class Sample21 extends JPanel {      // 2, 3
  public void paintComponent(Graphics g) {  // 4
    g.setColor(new Color(255, 180, 99));    // 5
    g.fillOval(100, 50, 100, 100);          // 6
  }
  public static void main(String[] args) {  // 7
    JFrame app = new JFrame();              // 8
    app.add(new Sample21());                // 8
    app.setSize(400, 300);                  // 8
```

```
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 8
    app.setVisible(true);                    // 8
  }
}
```

例題 2-1 の説明

十数行ほどのプログラムですが、最初なのでゆっくり説明しましょう。[32]

1. 冒頭の「`import ...;`」というのは、ライブラリにあるクラスを使いますよ、という指定です。これは、窓を出して絵を表示するために、そのような機能を持つオブジェクトを利用するからです。この 2 行は当面どのプログラムでも同じです。[33]
2. Java のプログラムはクラス定義の集まりです。クラス定義については最初の節で説明しましたが、ここでは先頭に **public** とあるので、クラスは**公開クラス**になります。[34] クラス名は「`Sample21`」です。[35] Java では公開クラス 1 つにつき 1 つのファイルに格納し、しかもそのファイル名はクラス名の後ろに拡張子「`.java`」を付けたものにする約束です。ですから、このプログラムを保存するファイル名は必ず `Sample21.java` としなければなりませんし、クラス名を変更したらファイル名も変更する必要があります。
3. クラス名の後に「`extends JPanel`」を指定しているので、作成するクラスは `JPanel` というクラスのサブクラスになります。`JPanel` は窓の内容となる四角い領域を表します。これを継承して、機能の一部、具体的にはメソッド `paintComponent()` を差し替えることで、自分の描きたい絵を表示する領域を作るわけです。
4. 続いて、差し替えるメソッド `paintComponent()` の定義があります。このメソッドは公開メソッドで値を返さず、[36] 画面に様々なものを描く「ペン」に相当する、**Graphics** オブジェクトを引数として受け取ります。[37] パラメタの名前は `g` なので、受け取ったオブジェクトはこの名前で参照します。そして、画面を描く必要が生じると `paintComponent()` が呼ばれるので、その中に記述した動作が実行されます。つまり、この中に記述した動作によって画面が描かれるのです。
5. `Graphics` オブジェクトのメソッド `setColor()` を呼ぶことで、ペンの色を設定します。引数としては **Color** オブジェクトを指定します。`Color` オブジェクトは「`new Color(赤, 緑, 青)`」のコンストラクタで作り、三原色である赤、緑、青の強さをそれぞれ 0〜255 の範囲の整数で指定します。
6. `Graphics` オブジェクトのメソッド `fillOval(`X`, `Y`, 幅, 高さ)` を呼ぶことで、ペンで楕円を描きます。このメソッドは図 11 にあるように、楕円に外接する長方形を指定することで、楕円の位置と大きさを指定します。[38] このプログラムの場合は、長方形の左上隅の座標が $(100, 50)$、幅と高さはどちらも 100 です (ですから長方形は実際には正方形で、描かれる図形も円になります)。

32)「`//…`」は**コメント** (注釈) といい、実行には影響しないので、プログラムに見て分かるようにメモなどを書いておくのに使います。ここでは説明に使う番号を書くのに使っています。

33)1 行目は、**AWT** (Abstract Windowing Toolkit) という、描画用のクラスの入ったパッケージ、2 行目は **Swing** という、ウィンドウや GUI 部品の入ったパッケージのクラス群を指定しています。

34)Java ではクラスを**パッケージ**という単位でまとめて整理できますが、その時公開クラスだけが、パッケージの外から (別のパッケージから) 利用できるようになります。本書ではパッケージは作りませんが、プログラムとして作るクラスは公開クラスにする必要があります。

35)クラス名をはじめ、本書でこの先 Java で使う「名前」はすべて「英字 (A〜Z、a〜z) ではじまり、英字か数字だけが並んだもの」ということにします。

36)メソッドに指定する public や void などの意味は最初の t 節で説明しました。

37)`Graphics` オブジェクトを渡してくれるのは、窓などの動作をつかさどる実行時システムです。

38)座標ですが、X 座標の値は右に行くほど大きくなり (これは普通です)、Y 座標の値は下に行くほど大きくなります (これは普通とは反対です)。また、単位は**ピクセル**つまり「画面上の点の個数」になっています。

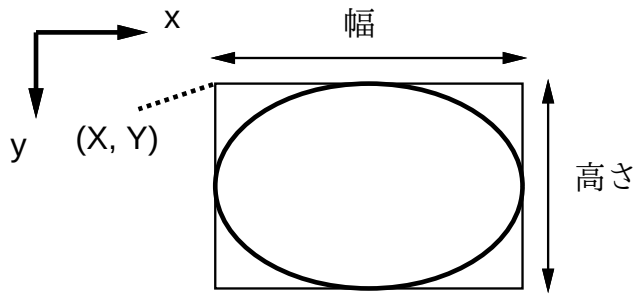


図 11: fillOval() による楕円の指定方法

7. プログラムとして実行されるようなクラスは、**main()** というクラスメソッドを持たなければなりません。プログラムの実行はこのメソッドから始まります。メソッド `main()` には、必ず次の指定が必要です。

   - `public` の指定 — `main()` は公開メソッドにします。
   - `static` の指定 — `main()` はクラスメソッドにします。
   - `void` の指定 — `main()` は何も値を返しません。
   - 引数 `String[] args` — `main()` は実行開始時に文字列の配列を引数として受け取ります。[39]

8. プログラムが実行開始した後の動作は、メソッド `main()` の中に記述します。ここでは次の内容です。[40]

   - **JFrame** は「窓」に対応するクラスであり、そのインスタンス (つまり窓オブジェクト) を作成する。以下で参照するため、この作成したオブジェクトは変数 `app` に入れる。[41]
   - 作成した窓の中に、クラス `Sample21` のインスタンスを生成してはめ込む。[42]
   - 窓の大きさを幅 400 ピクセル、高さ 300 ピクセルに設定。
   - 窓を「閉じる」操作があったらプログラムも終了するように設定。
   - 窓を「見える」状態にする (つまり開く)。

`setVisible(true)` の **true** とは「真 (はい)」に相当する値で、「偽 (いいえ)」に相当する **false** と組になって **boolean**(**論理型**) という型を構成しています。ここでは「はい」(見せる) を設定しているわけです。

説明が長くて、びっくりしましたか? しかし、プログラムを扱うということは、その 1 つひとつの記述がすべて意味がありますから、それが何をしているかを理解しておかないと、「ちょっと直して」みることすらできません。たった十数行ですので、説明と照合しながらじっくり読み返して、それぞれの記述の意味を納得しておいてください。

動かしてみよう!

ではいよいよ、最初のプログラムを動かしてみましょう。エディタ[43] を起動し、先に示したプログラムのコードを、その通りに打ち込んでくださ

39) 引数の名前は `args` でなくてもいいのですが、本書ではとくに変える必要もないので、常にこの名前を使います。

40) この部分 (窓を作成して中身を埋め込み表示) は、当面どのプログラムでも同じなので、だいたいこういうものだと思っておいて頂ければ大丈夫です。

41) 「型名 変数名 = 値;」で、変数を定義し、そこに値を格納します。型については後でまた説明しますが、クラスも型の一種です。

42) このクラスはまさに現在定義しているクラスであり、「絵を描く機能を追加した画面領域」だったことを思い出してください。なお、メソッド `main()` はこのクラスの中に (便宜上) 入っていますが、このクラスの機能を提供しているわけではなく、単に実行開始時の動作を指定するだけであることにも注意してください。

43) 「エディタ」というのはワープロソフトと異なり単純な文字だけを打ち込んだり修正するためのソフトで、「メモ帳」「SimpleText」「vi」「Emacs」などが代表的です。

い。[44] 打ち込み終わったら、必ず Sample21.java というファイル名で保存します。

では、Sample21.java の中のプログラムを実行させましょう。Java はコンパイル方式で実行するプログラミング言語です。**コンパイル**とは、Java 言語で書いたソースプログラムの命令の並びを、コンピュータの CPU が実行できる形のコードに変換することです。[45]コンパイルを行ってくれるプログラムのことを一般に**コンパイラ**と言います。ここで使うコンパイラのプログラムは javac という名前なので、javac コマンドに実行する Java のソースプログラムのファイル名を与えて、コンパイルします (図 12)。[46]

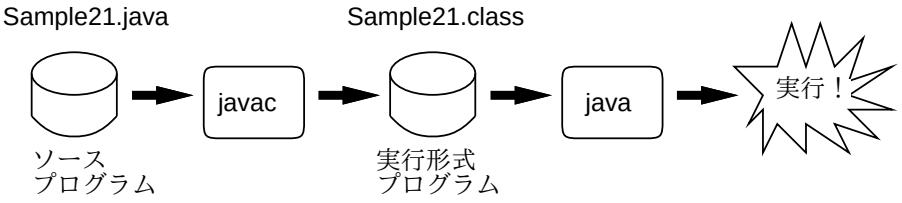


図 12: コンパイル方式と実行

```
% javac Sample21.java
```

プログラム中に検出可能なエラーがあると、コンパイル途中でエラーメッセージが出て止まりますので、エラーメッセージを参考にエディタでソースファイルを修正します (修正したらファイルを保存し直すのも忘れないようにしましょう)。再度コンパイルをして、エラーが出なくなるまで修正を繰り返します。この作業をコンパイルエラー取りといいます。コンパイルがうまくいくと、ソースファイルと同じ名前で、拡張子が.class のファイル (**クラスファイル**) ができます。たとえば Sample21.java をコンパイルした場合は Sample21.class になるはずです。[47]

クラスファイルはいよいよ、コンピュータで実行できます。実行は、java コマンドで行います。ただし java コマンドに与えるのはクラスファイル名ではなくて、実行するクラスのクラス名を与えます。ですから、例題 2-1 のプログラムを実行するは、次のようにクラス名 Sample21 を指定します。[48]

```
% java Sample21  ← クラス Sample21 の main() から実行開始
```

クラス Sample21 クラスを実行すると、先に定義した main() が実行され、図 10 の画面が現れるはずです。成功しましたか?

演習 2-1 例題 Sample21.java をそのまま打ち込んで動かしなさい。動いたら、プログラムを次のように手直ししてみなさい。

a. 円の色を変更する。また、円を楕円 (縦長/横長) に変更する。
b. 円を複数 (たとえば 3 個) 並べる。色は違う方がよい。
c. 円のかわりに正方形または長方形を描く。[49]
d. 色のコンストラクタで、new Color(赤, 緑, 青, α) のように 4 引数を取るものがある。α は「透明度」を表す 0〜255 の値。これを用いて、「半透明な 2 つ (以上) の図形が重なった」絵を描いてみる。

44)少しでも間違いがあるとエラーになりますが、エラーを出すのも経験のうちです。でも、エラーがあまり多いと大変ですから、やはり慎重に間違いがないように打ち込んでください。

45)Java 言語の場合は「ほんもの」の CPU ではなく **Java 仮想マシン**という仮想的な CPU のコードに翻訳します。皆さんは「Java のプログラムはさまざまなコンピュータの機種で動く」という話を聞いたことがあると思います。それは、さまざまな機種で動く Java 仮想マシンが用意されていて、Java 仮想マシンのコードに翻訳されたものをそれぞれのマシンで動かさせてくれるからできることなのです。

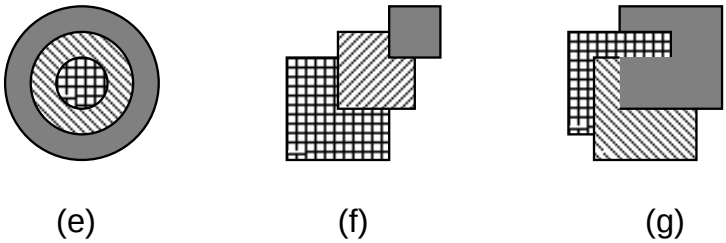
46)本書ではコンパイルも実行もコマンドを打ち込んで指示するやり方で説明していますが、メニューなどで指示するような環境を使っている人は適宜読みかえてください。本質的には同じことです。本書の例ではコマンドを打ち込む行は「%」ではじまっていますが、これも環境によって違います。

47)1 つのファイルに複数のクラスが入っている場合は、クラスファイルもそのクラスの数だけできます。今はまだクラスを 1 つしか入れていないのでクラスファイルも 1 つだけのはずです。

48)クラスを定義するファイルのファイル名と、その中で定義するクラス名は必ず同じものにしなければいけないので、必然的にそうなりますので、今のところあまり気にしなくて良いわけです。

49)fillOval() のかわりに fillRect(X, Y, 幅, 高さ) を使うことで、長方形を描くことができます。

e〜g. 図のように、3 つの図形が重なって見えるもの。[50]

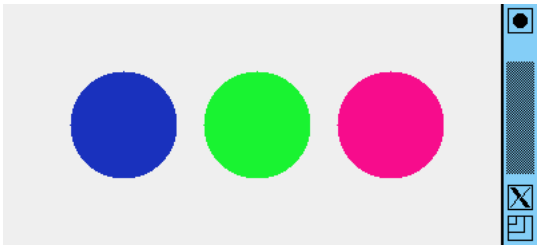


なお、描く絵の内容を変更するのは、メソッド `paintComponent()` の内側を変更するだけでできます。[51] 以下の例解でも、`paintComponent()` だけを示します。

例解 2-1-b

この問題は、(色を変える場合)1 つ目の円を描いて、それから 2 つ目の円を描く前に `setColor()` で色を別のものに変更することが必要になります。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(30, 50, 190));
  g.fillOval(50, 50, 80, 80);
  g.setColor(new Color(30, 240, 50));
  g.fillOval(150, 50, 80, 80);
  g.setColor(new Color(240, 15, 140));
  g.fillOval(250, 50, 80, 80);
}
```



例解 2-1-d

この問題は、透明度のある色を指定すること (α の値は小さいほど透明度が高くなります)、2 つの図形を半ば重なるように位置指定することが必要になります。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(30, 50, 190, 150));
  g.fillOval(100, 50, 100, 100);
  g.setColor(new Color(240, 15, 140, 150));
  g.fillOval(150, 50, 100, 100);
}
```

50) `fillOval()` 等で描くと、それまでに描いてあるものの「上に」描かれます。

51) あとの部分は変更しないでよいのですが、クラス名を変更した場合は (1) ファイル名を変更する、(2)`main()` の中の「`new Sample21()`」の部分を変更する、の 2 点を忘れなようにしてください。

例解 2-1-e

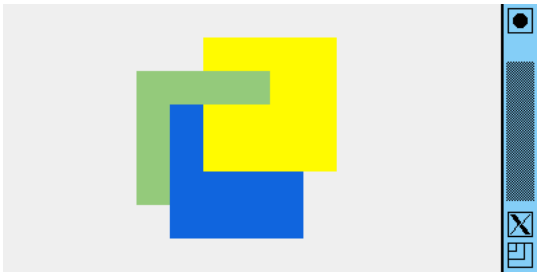
この問題は、「同心円」にするためには円の中心と半径をまず計算して、それに基づいて外接する正方形の左上隅と幅、高さを計算することがポイントになるでしょう。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(150, 200, 120));
  g.fillOval(100, 50, 100, 100);
  g.setColor(new Color(20, 100, 220));
  g.fillOval(120, 70, 60, 60);
  g.setColor(new Color(250, 250, 0));
  g.fillOval(140, 90, 20, 20);
}
```

例解 2-1-g

この問題は、3 つの正方形が「3 すくみ」になっていてどれが一番上か分からないように見えますが、実は一番下 (上) の正方形は 2 つに分けて描いてあることがポイントでしょう。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(250, 250, 0));
  g.fillRect(150, 25, 100, 100);
  g.setColor(new Color(150, 200, 120));
  g.fillRect(100, 50, 100, 100);
  g.setColor(new Color(20, 100, 220));
  g.fillRect(125, 75, 100, 100);
  g.setColor(new Color(250, 250, 0));
  g.fillRect(150, 75, 75, 50);
}
```



7　オブジェクトを調べる — API ドキュメント

先のプログラムはぶじ動きましたが、なんだか「こんなメソッドがあります」「あんなメソッドもあります」ばっかり言われているようで釈然としないだろうと思います。

Java では、どのようなクラスがあってそれらがどのようなメソッド等を持っているか、という情報をまとめたドキュメントのことを **API ドキュメ**

ントと呼んでいます。API ドキュメントは WWW 経由で Sun のサイトにあるものを見ることができますし、[52] ファイル一式を取り寄せて来て手元のマシン上に置いてブラウザで見ることもできます。

Java で主体的に (自分で書きたいと思うように) プログラムが書けるようになるためには、具体的にどのようなオブジェクトがあり、それらがどのようなメソッドを持っているのかが、自分で調べられるうになる必要があります。この先しばらくは、実際に API ドキュメントを開いて調べながら読んでください。もっとさまざまな図形を描くために、`Graphics` オブジェクトの機能を調べてみましょう。

まず、API ドキュメントは複数の「パッケージ」と呼ばれる単位に分かれています。パッケージを指定してからクラスを指定するのが普通ですが、パッケージが分からないときはクラス一覧だけから探してもよいです。

`Graphics` はパッケージ名からフルに書くと `java.awt.Graphics` で、つまり `java.awt` パッケージに含まれますから、左上の領域でそのパッケージを選択して、その後左下の領域で `Graphics` を選んでみてください。図 13 のような表示が得られるはずです。

図 13: API ドキュメントの表示

ここで、冒頭の説明は後で眺めて頂くことにして (色々難しいことが書いてありますから、ざっとでよいです)、少し下にある「コンストラクタの概要」「メソッドの概要」のところが大切です。API ドキュメントの一番多い使い方は、どのクラスについても、これらの部分を参照して、各種機能の呼び出し方を確認することです。たとえば一覧の中から `fillOval()` のところを選ぶと、このメソッドの機能とパラメタの説明を読むことができます (もちろん、既に学んだ通りのことが書かれているわけですが)。[53]

[52] `http://java.sun.com/javase/ja/6/docs/ja/api/`

[53] `Graphics` のメソッドには、冒頭に `abstract` と書かれているものが多くあります。これは、メソッドの呼び出し方はこのクラスで定めるけれど、具体的なメソッドはサブクラスで定義し直しますよ、という印です。このようなメソッドを持つ場合はクラス自体も `abstract` と指定します (実際 `Graphics` はそのようになっています)。

API ドキュメントを見ると、非常に多数のクラスやメソッドがあるので、圧倒されるかも知れません。しかし実際に必要なのは、「使いたいもの」に相当するクラスと、その「使いたい機能」に相当するメソッドを調べることだけです。具体的に、どんなクラスのどんなメソッドかって? それは、本書でこれから個別にご案内します。

ですから、本書で新しいクラスやメソッドが出て来たら、面倒くさがらずに API ドキュメントで確認する習慣をつけてください。[54] そうすれば次第に API ドキュメントに慣れて、本書で説明されていないものでも自分で調べて活用できるようになるはずです。Java を使いこなすというのは、最終的にはそういうことも含まれるわけです。

54)あるクラスにあるはずのメソッドの説明を探そうとして見つからない場合は、それが継承されたメソッドの一覧にないかどうかチェックしてみてください。

演習 2-2 API ドキュメントを表示し、ここまでに使ったクラスやメソッドを表示させてみなさい。少し慣れて来たら、次のことをやってみなさい。

a. クラス `java.awt.Color` の API ドキュメントを調べ、クラス変数 (実際には変更できないので定数になります) として、あらかじめ用意されている「標準的な色」に何があるか調べなさい。また、実際にプログラムで使ってみなさい。

b. クラス `java.awt.Graphics` の API ドキュメントを調べ、これまでに出て来た `fillOval()`、`fillRect()` のほかに、どのような図形を描くメソッドがあるかを調べてみなさい。また、興味を持ったものがあれば、それを描くプログラムを作ってみなさい。[55]

c. クラス `javax.swing.JFrame` の API ドキュメントを調べ、コンストラクタに文字列を 1 個指定すると、どのような効果があるのか調べなさい。また、実際に指定してみて確認しなさい。[56]

55)おすすめは、`fillArc()`、`fillRoundRect()`, `fill3DRect()`、`drawLine()` などでしょうか。「fill なんとか」の代わりに「draw なんとか」を使うと、塗りつぶすのでなく輪郭だけ描きます。また、`boolean` と指定されたパラメタには `true` または `false` を指定してみてください。

56)Java では文字列は`"..."`のように、ダブルクォート文字「`"`」で囲んだ中に文字を並べて書くことで指定します。

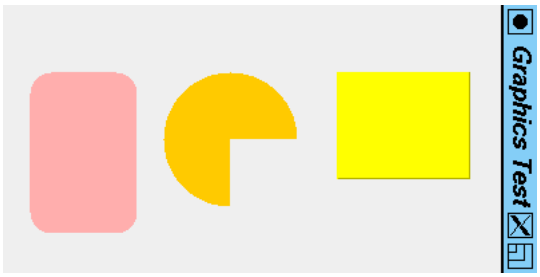
例解 2-2-abc

JFrame のコンストラクタに文字列を渡すとそれは窓のタイトルになります。API ドキュメントで確認できましたか。次に、`Color.PINK` 等のクラス変数を指定することで、一部の色は簡単に指定できます (ただし、どぎつい色が多いので、これまでの中間色の方が普通は好ましいです)。あと、ここでは角の丸い四角形、扇型 (角度 0 から 270 度まで)、浮き出て見える四角形を描いてみました。

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

public class ex22abc extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.PINK);
    g.fillRoundRect(20, 50, 80, 120, 30, 30);
    g.setColor(Color.ORANGE);
    g.fillArc(120, 50, 100, 100, 0, 270);
    g.setColor(Color.YELLOW);
    g.fill3DRect(250, 50, 100, 80, true);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame("Graphics Test");
    app.add(new ex22abc());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    app.setVisible(true);
  }
}
```



8　変数を使う、計算する

最初の例題が「円を 1 つ描く」だったので、今度は「円を 4 つ、等間隔で横に並べて描く」ことにします。そんなのは既にできるって? そうなのですが、まあ聞いてください。

等間隔に並べるということは、その円の X 座標がある決まった値ずつ大きくなっている、ということになります。その計算を暗算で (または電卓で) やってもいいのですが、せっかくのコンピュータなのですから、プログラムの中で計算させてみましょう。そのために、**変数**を使います。最初の節でも説明しましたが、変数は「値を入れておけるいれもの」です。次のようにすることで、**整数** (**int** 型) の値を入れておける変数 x を用意し、初期値として 100 を入れることができます。

```
int x = 100;
```

さて、位置 (正確には囲む矩形の左端) が $(x, 80)$、直径が 40 の円を描くとすると、この変数 x を参照して、次のようになります。

```
g.fillOval(x, 80, 40, 40);
```

次に、x の値を 60 だけ増やします (次の円を少し右に描くために)。[57]

```
x = x + 60;
```

びっくりしましたか? 数学では「x と x+60 が等しい」ことはあり得ませんが、これはそういう意味ではなく、単に「x+60 を計算し、その結果を再び x に格納する」という意味になります。つまり Java では「=」は「等しい」という意味ではなく、値を**代入**する (入れる)、という意味になります。[58]

話を戻すと、これで x が 60 増えたので、次はさっきと同じに `fillOval()` を呼び出せばずれた位置に円が描けます。このようにして、4 つの円を描くプログラムは次のようになりました (画面を図 14 に示します)。

57)足し算と引き算は「+」「-」で表します。何を当り前のことを、と思うかも知れませんが、掛け算は「*」、割り算は「/」なので全部同じというわけではないのです。あと、「割った余り」を求める演算「%」もあります。1 行に書く必要があるので、適宜かっこを使って「(a + b) / 2」などのように書いてください。

58)ちなみに「等しい」は「==」(イコール 2 つ) で表します。混乱しやすいので、注意してください。

例題 2-2: 4 つの円を描く

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

public class Sample22 extends JPanel {
```

```
  public void paintComponent(Graphics g) {
    g.setColor(new Color(220, 70, 250));
    int x = 100;
    g.fillOval(x, 80, 40, 40);
    x = x + 60;
    g.fillOval(x, 80, 40, 40);
    x = x + 60;
    g.fillOval(x, 80, 40, 40);
    x = x + 60;
    g.fillOval(x, 80, 40, 40);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample22());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

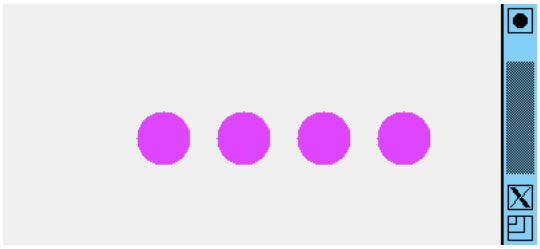


図 14: 4 つの円を描くプログラムの画面

ここで、プログラムの重要な原理について注意しておきましょう。それは、メソッドの中に書かれたプログラムは原則として「上から順に」実行されていく、ということです。つまり、「x に 100 を入れ、円を描き、x を 60 増やし、円を描き、x を 60 増やし、…」のように続いていくわけです。[59)]

演習 2-3 例題 Sample22.java をそのまま打ち込んで動かしなさい。動いたら、プログラムを次のように手直ししてみなさい。

a. 円の数をもっと多くする (5 とか 6 とか)。

b. 水平でなく、少し傾いて (右に行くほど円の位置が上になるように) 並べる。

c. 今は全部の円が同じ色ですが、右に行くほど徐々に色が変化するようにする (変化のしかたはお任せします)。

d. 円の配置を完全に規則的でなく、「3 番目の円だけ少し上にずれている」ようにする。

59)実は、ここまでにやった演習問題でも、「後から塗った色は前の色を上描きする」ことを使っていましたから、上から順番に実行することは利用していたわけです。でも大切なことなので、ここで改めて注意させていただきました。

例解 2-3-bcd

b や c は要するに、Y 座標や色の生成に使う値も変数にして変化させて行けばよいわけです。このために、変数 y と red を追加しました。[60][61]

60)この例のように、同じ型の変数は 1 つの宣言でまとめて定義できます。

61)変数が red だけなのは、色の赤の成分だけ変化させることにしたためです。もちろん、青や緑の成分も変化させるようにしてもかまいません。

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

public class ex23bcd extends JPanel {
  public void paintComponent(Graphics g) {
    int x = 100, y = 120, red = 220;
    g.setColor(new Color(red, 70, 250));
    g.fillOval(x, y, 40, 40);
    x = x + 60; y = y - 20; red = red - 50;
    g.setColor(new Color(red, 70, 250));
    g.fillOval(x, y, 40, 40);
    x = x + 60; y = y - 20; red = red - 50;
    g.setColor(new Color(red, 70, 250));
    g.fillOval(x-10, y-30, 40, 40);
    x = x + 60; y = y - 20; red = red - 50;
    g.setColor(new Color(red, 70, 250));
    g.fillOval(x, y, 40, 40);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new ex23bcd());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

なお、c の「3 番目だけは少しずらす」は、その円を描くところだけちょっと x や y から値を引いてずらしています。

9　付録: Java 言語のデータの種類と型

最初のプログラムはどうでしたか。ここでは、Java 言語で使えるさまざまなデータの種類と型について、説明しておきましょう。[62]

62)付録では、Java 言語のさまざまな機能について整理して説明して行きます。本文で細かい説明をしはじめるとプログラムの説明とごちゃごちゃになって大変なので、このようにしました。

Java ではデータは大きく分けて「オブジェクト」と「値」に大別されます。オブジェクトは最初に学んだように、クラスによって定義されていて、さまざまな機能を提供してくれます。ここでも JFrame、JPanel、Graphics などさまざまなクラスのオブジェクトが出てきました。そして、オブジェクトに対する操作はメソッドを呼び出すことで行いましたね。

```
JFrame app = new JFrame();    ←生成
app.add(...);                 ←操作 (メソッド呼び出し)
```

「型 (type)」というのは、プログラミング言語用語で「データの種類」という意味です。オブジェクトの種類はそのオブジェクトがどのクラスのインスタンスかで決まりますから、オブジェクトについてはクラスが型に対応します。

表 1: Java 言語の基本型

基本型	内容	リテラルの例	対応するクラス
byte	8 ビット整数	—	Byte
short	短精度整数	—	Short
int	整数	3, -10	Integer
long	長精度整数	100L, -5l	Long
float	短精度実数	3.14F, -5.0e+10f	Float
double	倍精度実数	3.14, 1.0e-8	Double
boolean	論理値	true, false	Boolean
char	文字	'a', 'b'	Character
—	文字列	"abc", "0"	String

一方、「値」というのは数値のように計算したり大小比較したりするような基本的な (内部構造を持たない) データを言います。「値」に対応する型のことを**基本型**と呼びます。

Java には表 1 にあるように 8 種類の基本型があります。主に使うのは整数 int、倍精度実数 double、論理値 boolean などでしょう。。Java では次の 2 種類の実数が使えます。

- double — 64 ビット IEEE754 形式浮動小数点。小数点のついた定数 (例: 3.14) や指数記号 e のついた定数 (例: 3.0e+12 — 3.0×10^{12} の意味) はこの型になる。
- float — 32 ビット IEEE754 形式浮動小数点。定数は double の定数の後ろに「f」をつけたもの (例: 3.14f、3.0e+12f など)。

もちろんビット数の多い方が精度がよいので、当面は精度のよい double の方を主に使うことにします。実世界では整数は実数の部分集合ですが、コンピュータでは整数と実数は全く別のもので、必要に応じて相互に変換することに注意してください。

基本型の多くは、プログラム中にじかに値 (定数) を書くことができます。これを**リテラル** と言います。int と long、double と float などはリテラルの書き方は同様になるので、最後に「l」や「f」をつけることで後者 (頻繁には使わない方) を現すようになっています。実数の型については、「e」に続けて指数を指定する記法も用意されています。たとえば「1.0e+20」は「1.0×10^{20}」を現すわけです。

表の最後に文字列が載っていますが、文字列だけは基本型ではなく、クラス String のオブジェクトです。[63] ですが、文字列のリテラルが書けないと不便なので、クラス String に限っては例外的にリテラルが書けるようになっています。シングルクォート「''」で囲んだ文字 (1 文字だけ) は char 型で、ダブルクォート「""」で囲んだ文字列 (0 文字以上何文字でも) は String オブジェクトなので、混同しないように注意してください。

63)中に複数の文字が入っているなど、複雑な内部構造を持つため、このようになっています。

Java ではさまざまな場面についてオブジェクトを使うので、8 つの基本型についてもオブジェクトとして扱うための、「対応するクラス」が用意されています。これを**包囲クラス** (wrapper class) と呼んでいます。包囲クラスのインスタンスは、その内部に対応する基本型の値を保持するようなオブジェクトになっています。

包囲クラス型の変数に対応する基本型を代入したり、逆に基本型の変数に対応する包囲クラスを代入するときは、これらの間で自動的に変換を行ってくれます。これを自動**ボクシング** (boxing)/**アンボクシング** (unboxing) と呼びます (図 15)。[64]

64)「箱に入れる」「箱から出す」という意味です。

```
Integer i1 = 100;  // boxing。new Integer(100) と同等。
int k1 = i1;       // unboxing。i1.intValue() と同等。
```

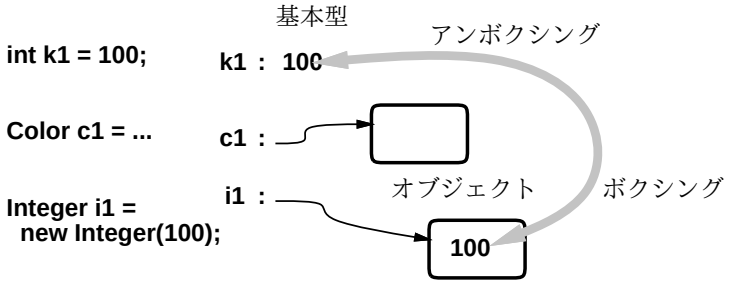


図 15: 包囲クラス型とボクシング/アンボクシング

これらに加えて、異なる数値型どうしの演算や代入は、より「広い」(表 1 で下の方にある) 数値型への変換は自動的に行われます。逆向きの (より「狭い」数値型への) 変換は「(型名) 式」という型変換指定 (**キャスト**と呼びます) を書く必要があります。この場合、狭い数値型で現せない部分は切捨てられます。たとえば実数を整数にキャストすると、小数点以下は切捨てられて整数に変換されます。

```
long l1 = 100 + 20L;  // 100 は long に変換して演算
float f1 = 1.0f + l1; // l1 は float に変換して演算
double d1 = 3.14 + f1 // f1 は double に変換して演算
int i1 = (int)d1;     // d1 を int にキャスト
```

基本型に対しては、表 2 の演算子が使えます。もちろん、演算によって使える型の値が決まっていて、増減や加減乗除などは数値の型、ビット操作は整数をビットの列と見なして行うので整数の型のみに使え、論理値演算は論理値型だけに使えます。なお、この表で並んでいる順番は「結び付きの強い順」になっています。たとえば、(かっこを使って順番を指定しない場合) 乗除算は加減算よりも先に演算されるわけです。[65]

65)さらに、表には掲載しませんでしたが、代入と同じ強さの演算子として`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`^=`、`|=`、`<<=`、`>>=`、`>>>=`があります。たとえば「`x += 1`」は「足し込む」演算であり、「`x = x + 1`」と同じ結果をもたらします。他も同様です。

```
double x = 10.0 + x * 3.14;  // 乗算を先に実行
```

なお、代入「`=`」も演算子の仲間で、代入した値を結果とします。このため、「`x = y = 0`」のような式も問題なく書くことができます。

表 2 にあるような演算はオブジェクトの型に対しては使うことができません。オブジェクトに対する操作はあくまでもメソッドを呼び出すことが原則

表 2: Java 言語の演算子

演算子	意味
++, --, +, -, ~, !	1 増す/減らす、正/負符号、ビット/論理値反転
*, /, %	乗算、除算、剰余
+, -	加算、減算
<<, >>, >>>	ビットシフト (左、算術右、論理右)
<, >, <=, >=	小さい、大きい、以下、以上
==, !=	等しい、等しくない
&	ビット単位の AND
^	ビット単位の XOR
|	ビット単位の OR
&&	論理値の「かつ」
||	論理値の「または」
=	代入

です。例外として、「==」と「!=」を使って 2 つのオブジェクトが同一のオブジェクトかどうかを調べることだけは可能です。ただし、オブジェクトとして同一かどうかというのは、オブジェクトの中に入っている値が同じかどうかとは別の問題なので注意が必要です。

```
new Integer(10) == new Integer(10)  // false
"abcdef" == "abcdef"    // true か false か不明
```

上の例は、2 回 Integer オブジェクトを生成しているので、これらは必ず別のオブジェクトであり、「==」による比較結果は「いいえ」になります。下の例は、同じ内容の文字列が出て来たときに 1 つの String オブジェクトで済ませるか別々のオブジェクトにするかは、実行系によって違っている可能性があります。

変数は定義すると同時に初期値を入れるのがおすすめですが、初期値を何も入れない場合、数値型の変数では 0(に対応する値)、論理値型では false、オブジェクト型の変数については「何のオブジェクトも入っていない」ことを表す **null** という特別な値が入った状態になります。

10 while 文による繰り返し

前節まででは「メソッドの動作は記述されたコードを上から順番に実行する」ことを学びました。しかし場合によっては、コードの一部を繰り返し実行したり、ある部分を実行しないなどの処理が必要なこともあります。以下では引き続き「絵を描く」ことを題材に、これらの方法について学んで行きます。

前節の「並んだ円を描く」プログラムを思い返してみてください。その中枢部分は次のような形をしていました。

```
…
g.fillOval(x, 80, 40, 40);
x = x + 60;
```

```
g.fillOval(x, 80, 40, 40);
x = x + 60;
…
```

つまり、まったく同じ 2 つの行を「繰り返して」いました。このように、同じ内容を繰り返し書くやり方には、次のような弱点があります。

- プログラムが長くなり、書くのも修正するのも大変になる。
- 繰り返す回数があらかじめ決まっている必要がある。

こうする代わりに次のように直すことで、この弱点を解消できます。

```
…
while(…条件…) {
  g.fillOval(x, 80, 40, 40);
  x = x + 60;
}
…
```

この書き方のことを、**while 文**と呼びます。[66] while 文は一般に、次のような形をしています。

```
while(条件) {
   本体…
}
```

これは、「本体…」と記された部分を何回も繰り返し実行することを意味します。つまりこうすることで、先の例でいえば繰り返される 2 行は 1 度ずつ書くだけで、while 文の機能によって繰り返し実行されるわけです。問題は「いつまで」繰り返すかということですが、それは丸かっこの内側に指定する「条件」が成り立つ間繰り返し実行する、ということになっています。では、円を並べる例題で見てみましょう。[67]

66) 繰り返しのことを**ループ**と呼ぶので、while ループという言い方をすることもあります。

67) **注意:** 繰り返しを使うと、プログラムが間違っていた時に「永遠に終わらない」ことが起こります。そのような場合は、java コマンドを打ち込んだ窓で「Control-C」(Ctrl キーを押しながら「C」のキーを打つ) でプログラムを停止させられます。覚えておきましょう。

例題 3-1: 円を繰り返し描く

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

public class Sample31 extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(new Color(220, 70, 250));
    int x = 100;
    while(x < getWidth()) {
      g.fillOval(x, 80, 40, 40);
      x = x + 60;
    }
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
```

```
    app.add(new Sample31());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

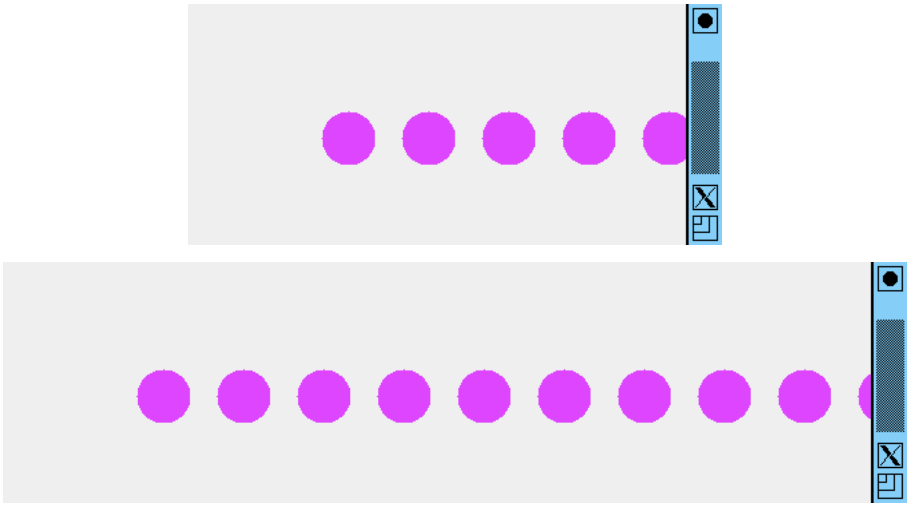
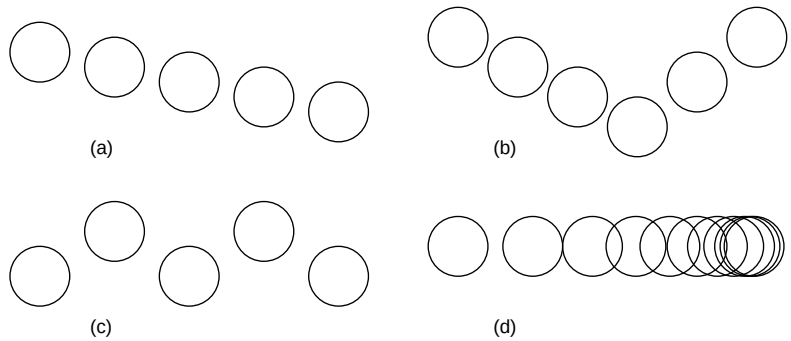


図 16: 繰り返しを使って円の並びを描く

このプログラムは前章末から上にかけて説明したように、変数 x を繰り返し増やしながら円を描くことで、円を横に規則的に並べます。そして繰り返しの条件ですが、「x がこのパネルの幅より小さい間」繰り返すので、[68] 窓の幅一杯に円が並びます。また、窓を横にどれだけ伸ばしても、円の列はそこまで伸びて見えます (図 16)。

68)メソッド getWidth() は、このオブジェクトつまりパネルの幅を返すメソッドです。同様に getHeight() は高さを返します。

演習 3-1 例題 Sample31.java を打ち込んでそのまま動かしてみなさい。動いたら、プログラムを次のように手直ししてみなさい。



a. 円が水平でなく、斜めに並ぶようにする。[69]

b. 円が途中まで下がって行き、その後上って行くようにする。[70]

c. 円が水平でなく、「上下にジグザグに」並ぶようにする。[71]

d. 円の間隔が徐々に狭くなって行くようにする。[72]

69)ヒント: 縦の位置を変数 y に入れ、徐々に増やす。

70)ヒント: while 文を 2 つ使って下がって行くのと上がって行くのに対応させる。

71)ヒント: y の計算式を 2 つの値が交互に出て来るように工夫。

72)ヒント: x の間隔を一定値でなくする。具体的には別の変数を用意して、この値を間隔として、間隔を徐々に減らす。

最後の問題 d では、「徐々に」狭くするために「小数点つきの数 (たとえば 0.8)」を間隔に掛けて行くとよいでしょう。そうすると、間隔は小数点つきの数つまり double 型の変数ということになります。しかし、位置を示す変数 x は int 型なので、位置に間隔を足すところで間隔の値を int 型にキャストしてください。[73]以下に例解を示しますが、これらもメソッド paintComponent() だけを掲載します。他の部分は例題と同じです。

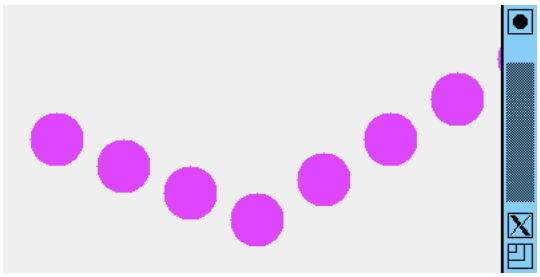
73)そうしないと、前章付録で解説したように、足した結果が実数型になってしまい、x に入れられなくはります。

例解 3-1-b

この問題は、途中まで「下がって行く」while 文を使い、その後「上がって行く」while 文を使うようにします。「途中まで」ということは、条件として x の値を「途中」に相当する値と比較することで実現できます。[74]

74)なお、ここでやっているように、1 つの「int …」(変数宣言) で、2 つ以上の変数をまとめて宣言して初期値を入れることもできます。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(220, 70, 250));
  int x = 20, y = 80;
  while(x < 150) {
    g.fillOval(x, y, 40, 40);
    x = x + 50; y = y + 20;
  }
  while(x < getWidth()) {
    g.fillOval(x, y, 40, 40);
    x = x + 50; y = y - 30;
  }
}
```



例解 3-1-c

この問題は簡単に解説します。たとえば、y を最初 80 に初期化したとします。続いて、次の文を実行するとどうでしょうか。

```
y = 200 - y;
```

200 − 80 == 120 ですから、y の値は 120 になります。続いて再度同じ文を実行すると、200 − 120 == 80 ですから、もとの 80 に戻ります。ですから、ループの中にこの処理を入れておくことで、2 つの高さの円が交互に現れるようにできます。

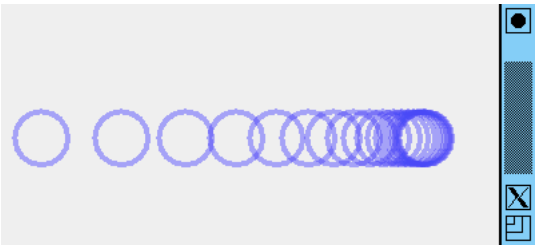
例解 3-1-d

この問題は、x を増やす量を別の変数 dx に入れておいて、この値を徐々に小さくしていくことで実現します。上で解説したように、dx は double 型にしておいて、それを x に足す時には「(int)dx」という書き方 (キャスト) で整数に変換します。[75][76]

75)dx が 1 未満になると、それ以上位置が変化しなくなりますから、while 文の条件も「dx >= 1.0」のようにする必要があることに注意してください。

76)「>=」は「以上」を表します。比較演算子としては「==」(等しい)、「!=」(等しくない)、「<」(より小)、<=」(以下)、「>」(より大)、「>=」(以上) の 6 つが使えます。

```
public void paintComponent(Graphics g) {
  int x = 10;
  double dx = 60.0;
  g.setColor(new Color(70, 70, 240, 120));
  ((Graphics2D)g).setStroke(new BasicStroke(4));
  while(dx >= 1.00) {
    g.drawOval(x, 80, 40, 40);
    x = x + (int)dx;
    dx = dx * 0.8;
  }
}
```



dx は毎回 0.8 倍することで減らしていきます。また、円が重なっているようすが分かりやすいように、fillOval() の代わりに drawOval() を使って輪郭だけを描くようにしています。このとき、線の太さを太くしたいので、次のようにして「4」に設定しています。

- パラメタ g を Graphics2D にキャストする。[77]
- キャストしたオブジェクトに対して、メソッド setStroke() を呼んで太さを設定する。[78]

77)キャストできるオブジェクトのが渡されて来るようになっています。キャストする理由は、メソッド setStroke() は Graphics2D オブジェクトが持つメソッドだからです。

78)パラメタとして「太さ 4 の単純線」を表すオブジェクトである new BasicStroke(4) を渡すことで太さを設定しています。

11 計数ループと配列

前節の演習問題では、いく通りかの制御方法でループを使って来ましたが、実はもっともよく使うやり方は、「1, 2, … N」のようにある変数を 1 ずつある値まで増やして行くものです。[79]このようなループのことを「数えて行く」ことから**計数ループ**と呼びます。また、計数ループに使う変数のことを**カウンタ**とも呼びます。計数ループを while 文で書いてみましょう。

```
int i = 1;          ←カウンタの初期化
while(i <= n) {     ←繰り返しを続ける条件
  繰り返しの処理…
  ++i;              ←カウンタの更新 (1 増やす)
}
```

このような「初期化」「条件」「更新」を別々に書いてあるより、まとめて書いてあった方が見やすく間違えにくいので、[80]以下に示すような **for 文**としてまとめて書けるようになっています。[81]

```
for(int i = 1; i <= n; ++i) {
  繰り返しの処理…
}
```

79)なお、「++i」というのは「i = i + 1」と同じ意味で、1 増やしたり減らしたりすることはとても多いので短く書けるようになっています。もちろん 1 減らすのは「--i」です。

79)とくに「更新」を書き落とすと「永遠に止まらない」ループになってしまいますが、「更新」は本体の処理の後に書くので忘れやすいのです。なお、止まらなくなった時は java コマンドを打った窓で Control+C を打つのでしたね

81)Java 言語の for 文は、ここで説明しているように while 文の書き方を「まとめた」だけなので、while 文と同様にさまざまな繰り返しの制御ができますが、本書では分かりやすさのため、原則として計数ループの時にだけ for 文を使うようにします。

あと、上の説明では「1 から」数えていましたが、コンピュータでは「0 から N の手前まで」、つまり「0, 1, 2, …, N-1」のように数えることが多くあります。この場合、for 文はのようになります。

```
for(int i = 0; i < n; ++i) {
  繰り返しの処理…
}
```

計数ループは分かりやすく使いやすいので、これから頻繁に登場します。たとえば、先にやった「円を配置する」問題も、(今更ですみませんが) 次のように考えた方が分かりやすいかも知れません。

- 計数ループで i を 0, 1, … $N-1$ と変化させる。
- i から計算式で x や y などの値を計算して使用する。

ここで、計数ループと同時に使われることが多い**配列**という機能を説明しておきます。配列とは「決まった個数の変数が並んだもの」であり、その並んだものの中で「何番目」かをそのつど指定して使用します (図 17)。[82]

82)数学でいうと「添字のついた変数」(x_0, x_1, ...) のようなものだと思ってください。

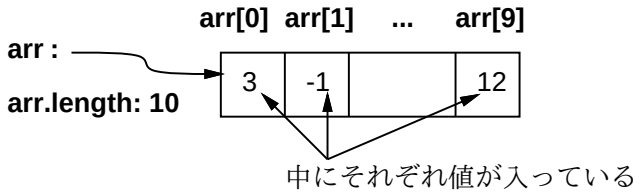


図 17: 配列の概念

配列は変数が並んだものですから、個々の変数の型によって「整数が並んだ配列」「文字列が並んだ配列」などさまざまなものが作られます。このため配列の型は、要素の型のうしろに「[]」をつけて表します。そして、配列を実際に使う時は、配列もオブジェクトなので new 演算子を使って作りますが、そのとき「いくつ並んでいるか」を指定します。

```
int[] arr = new int[10];  // int が 10 個
String[] names = new String[100]; // 文字列が 100 個
```

そして、個々の値を入れたり参照するときには次のように変数名に続いて [] の中に「何番目」を指定する式を書きます。[83]

83)このために、配列を表すのに「[]」を使っていたのですね。なお、[] の中に書く何番目かを表す指定のことを**添字**と呼びます。添字は 0 から始まるので、要素数が 10 個なら添字は 0〜9 までの範囲が指定できます。範囲を外れた添字を指定すると実行時に例外 (エラー) が発生します。

```
arr[5] = 100;
g.drawString(names[i+1]);
```

配列の要素数は.length というフィールドを参照することで取り出せます。たとえばここの例では arr.length は 10 です。

配列と計数ループはどのように組み合わせられるのでしょう? たとえば、配列 arr のすべての要素に-1 を入れるとしたら、次のような for 文を使います。これで、arr の 0 番目から最大の要素まで順番に-1 を入れていきます。

```
for(int i = 0; i < arr.length; ++i) arr[i] = -1;
```

例題 3-2: ジグザグを描く

これまで、円とか長方形とか簡単なメソッドだけで描ける図形を描いてきましたが、今度はもう少し込み入った図形を描いてみましょう。そのために、複数の点の座標を配列で受け取る、Graphics の次のメソッドを使います。

- `drawPolyline(`XS`,` YS`,` N`)` — 複数の点を結んだ折れ線を描く。
- `drawPolygon(`XS`,` YS`,` N`)` — 閉多角形の輪郭を描く。
- `fillPolygon(`XS`,` YS`,` N`)` — 閉多角形の内部を塗りつぶす。

これらはいずれも、N 個の X 座標と N 個の Y 座標をそれぞれ整数の配列で渡し、3 番目の引数として点の数を渡します。[84)]

84)折れ線と閉多角形の違いは、折れ線の最後の点と最初の点を結んだものが閉多角形ということです。

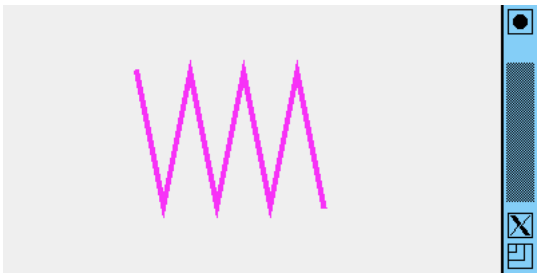


図 18: ジグザグ

では、これを使って「ジグザグ」を描いてみます (図 18)。点の数が 8 なので、大きさ 8 の配列 xs と ys を用意し、前者には for 文を使って等間隔の X 座標 (100, 120, ...) を入れています。[85)]後者は偶数番目が 50、奇数番目が 150 とすることで、全体がジグザグになります。[86)]

85)まず 0 番目に最初の値を入れ、1 番目からは for 文の中で「1 つ前より 20 多い」値を入れています。

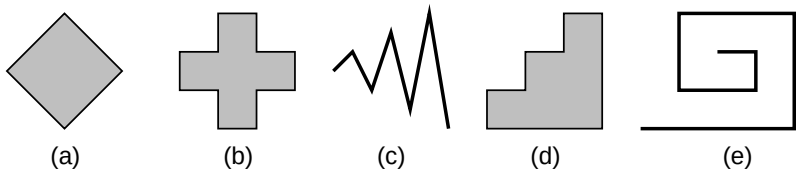
86)Java では「=(代入)」も演算子で、変数に入れたのと同じ値を返しますから、それをさらに次の変数に入れる、というふうにして複数の代入を次々に書いています。

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

public class Sample32 extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(new Color(240, 50, 240));
    ((Graphics2D)g).setStroke(new BasicStroke(4));
    int[] xs = new int[8], ys = new int[8];
    xs[0] = 100;
    for(int i = 1; i < 8; ++i) { xs[i] = xs[i-1] + 20; }
    ys[0] = ys[2] = ys[4] = ys[6] = 50;
    ys[1] = ys[3] = ys[5] = ys[7] = 150;
    g.drawPolyline(xs, ys, 8);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample32());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
```

```
    }
  }
```

演習 3-2 `Sample32.java` をそのまま打ち込んで動かしなさい。動いたら、下の a〜d のような図形を描いてみなさい。c〜e については、ループを使って折れ数を多くできるとなおよいでしょう。[87]

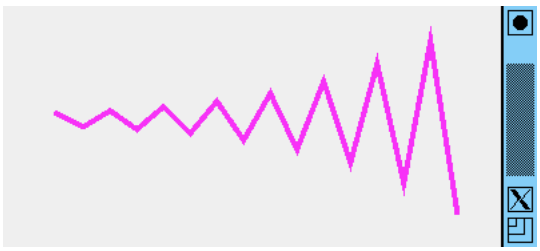


87)ヒント: d と e でループを使う場合は、0 番目に「最初の位置」を入れておき、その後は「1 番目と 2 番目」「3 番目と 4 番目」のように 2 つずつ点の位置を計算して行きます。そのためには、for ループでカウンタの更新を「`i = i + 2`」のようにして 2 つずつ増えるようにします。

例解 3-2-c

X 座標については、例題と同じに for ループを使って等間隔で並べます。Y 座標については、上下への移動量を `d` とし、1 つ前の点の値に `d` を足したものが次の点の値になります。`d` を毎回-1.2 倍することで、正と負が交互にあらわれ、絶対値が徐々に増えて行きます。[88]

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(240, 50, 240));
  ((Graphics2D)g).setStroke(new BasicStroke(4));
  int[] xs = new int[16], ys = new int[16];
  double d = 10.0;
  xs[0] = 40; ys[0] = 80;
  for(int i = 1; i < 16; ++i) {
    xs[i] = xs[i-1] + 20;
    ys[i] = ys[i-1] + (int)d;
    d = d * -1.2;
  }
  g.drawPolyline(xs, ys, 16);
}
```



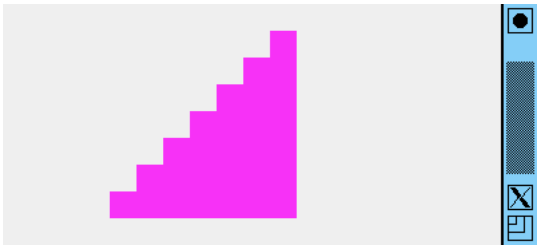
88)`d` は実数なので、先の例題と同じ理由から、整数に足す時に整数にキャストしています。

例解 3-2-d

ヒントにあるように、2 つずつ点の位置を計算しています。[89]最後に右下の点だけ特別扱いで設定します。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(240, 50, 240));
  int[] xs = new int[16], ys = new int[16];
  xs[0] = 80; ys[0] = 160;
  for(int i = 2; i < 16; i = i + 2) {
    xs[i-1] = xs[i-2]; xs[i] = xs[i-1] + 20;
    ys[i-1] = ys[i] = ys[i-2] - 20;
  }
  xs[15] = xs[14]; ys[15] = ys[0];
  g.fillPolygon(xs, ys, 16);
}
```

89)X 座標については、2 つのうち 1 つ目は 1 つ手前の点と同じ、次は 20 だけ増える。Y 座標については、2 つとも 1 つ手前の点より 20 増える。というふうになります。

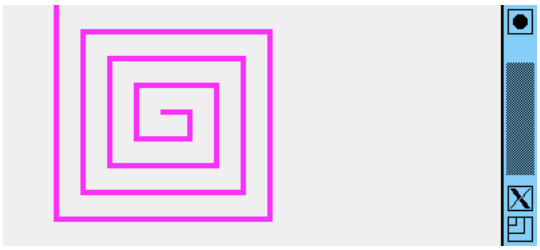


例解 3-2-d

内側から描くとして、うず巻きの線が 2 本ずつ同じ長さで、その前の 2 本より一定長いことと、その 2 本の位置は「X だけ増える/減る」「Y だけ増える/ 減る」で、増えると減るが交互になることに気がついたでしょうか。あとはこれまでの問題と同様です。[90)]

90)一定ずつ増えるのは変数 d を一定ずつ増やし、増える/減るが交互になるのは変数 sign を 1 回ごとに 1、-1、1、-1 と反転しながら掛け算することで扱っています。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(240, 50, 240));
  ((Graphics2D)g).setStroke(new BasicStroke(4));
  int[] xs = new int[17], ys = new int[17];
  xs[0] = 120; ys[0] = 80;
  int d = 20, sign = 1;
  for(int i = 2; i < 17; i = i + 2) {
    xs[i-1] = xs[i] = xs[i-2] + d*sign;
    ys[i-1] = ys[i-2]; ys[i] = ys[i-1] + d*sign;
    d = d + 20; sign = -sign;
  }
  g.drawPolyline(xs, ys, 17);
}
```



12　枝分かれ

ここまでで、さまざまな条件による繰り返しを扱って来ましたが、条件を使う別の機能として、「条件に基づいてある処理をやったりやらなかったりする」ものがあり、これを**枝分かれ**と呼びます。Java では条件は **if 文**によって指定します。その一番基本的な形は次のものです。

```
if(条件) {
    動作…
}
```

これは、「条件が成り立った時だけ動作を行う」ことを指定しています。実際にプログラムを書く場合には、「条件が成り立たなかった場合にはまた別の動作を行う」必要があることも多いですが、それには次のようにして「別の動作」を指定します。

```
if(条件) {
   動作…
} else {
   別の動作…
}
```

もう少し高度な枝分かれとして、「条件 1 が成り立ったら動作 1 を実行し、そうでなくて条件 2 が成り立ったら動作 2 を実行し、(以下同様…)、どの条件も成り立たなかったら別の動作を実行する」というものがあります。これは次のように書きます。

```
if(条件 1) {
   動作 1…
} else if(条件 2) {
   動作 2…
} else if(条件 3) {
   動作 3…
   :
} else {
   別の動作…
}
```

この形のことを **if-else の連鎖**と呼びます。その動作は、条件を上から順に調べて行って、最初に成り立った条件に対応する動作だけを実行する、というふうに考えるとよいでしょう。なお、「別の動作」を指定しなくてよい場合は、最後の `else` 以下を省略します。

例題 3-3: 枝分かれで動作を変化させる

ではさっそく枝分かれを使ってみることにして、先に挙げた円の並びの演習問題の「円が途中まで下がって行き、その後上がって行くようにする」を枝分かれでやってみましょう。[91)]

91)円の並びの例題は while ループを使っていましたが、こちらの版では計数ループを使って繰り返し、X 座標はカウンタの値をもとに計算するようにしてみました。

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

public class Sample33 extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(new Color(220, 70, 240));
    int y = 80, dy = 30;
    for(int i = 0; i < 20; ++i) {
      int x = 50 + i*50;
      g.fillOval(x, y, 40, 40);
      if(y+80 > getHeight()) { dy = -dy; }
      y = y + dy;
    }
  }
  public static void main(String[] args) {
```

```
        JFrame app = new JFrame();
        app.add(new Sample33());
        app.setSize(400, 300);
        app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        app.setVisible(true);
    }
}
```

このプログラムでは、y の値は最初 80 で、ループの中で dy ずつ増えて行きます。ただし、y+80 が窓の高さより大きくなったら、dy に-dy を入れることでマイナスの値にし、以後はマイナスの値を加えるのでその分だけ減るようになるわけです (図 19)。[92]

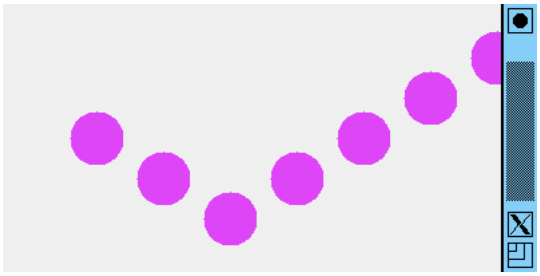


図 19: 枝分かれで動く方向を変更する

92)このプログラムを見ると、繰り返しの中に枝分かれが入っていることが分かります。逆に枝分かれの中に繰り返しを入れることもできますし、繰り返しの中に別の繰り返し入れることもできます。このように、枝分かれや繰り返しを自由に組み合わせることで、込み入った処理が記述できるわけです。

演習 3-3 例題 Sample33.java を打ち込んでそのまま動かしてみなさい。動いたら、プログラムを次のように手直ししてみなさい。

a. さらに、円が窓の上に出て行きそうになったら、また方向が代わって下がって行くようにする。[93]

b. 円を水平に並ぶようにして、ただし 3 番目の円だけが少し上に描かれるようにする。[94]

c. 円は水平に並ぶようにして、3 番目の円だけは赤、また 5 番目の円だけは青い色にする。[95]

d. その他、自分の好きなバリエーションで円を並べる。

93)ヒント: y が適当な値より小さくなった場合も、dy の符号を反転する。

94)ヒント: カウンタの番号が 2 のときが 3 番目。このときに少し上に描き、それ以外の場合は「通常の位置」に描くようにする。

95)ヒント: if-else の連鎖を使って、3 番目、5 番目、その他の場合に枝分かれする。

これらの問題をやるに当たって、既に学んだ大小比較の条件だけでなく、これらを組み合わせた「条件 1 と条件 2 のどちらかが成り立つ (条件 1 または条件 2)」「条件 1 と条件 2 の両方が成り立つ (条件 1 かつ条件 2)」などの**複合条件**を指定したいことがあるかも知れません。Java では次の 3 つが使えます。

- 条件 1 `||` 条件 2 — 「または」を表す。例: `x < 0 || x > 9` — 「x が 0 未満または 9 より大きい」。
- 条件 1 `&&` 条件 2 — 「かつ」を表す。例: `x >= 0 && x <= 9` — 「x が 0 以上かつ 9 以下」。
- `!`条件 — 「〜でない」を表す。例: `!(x % 2 == 0 || x % 3 == 0)` — 「x が 2 の倍数や 3 の倍数ではない」。[96]

最後の例のように、複雑な条件のときは適宜かっこを使ってください。

96)「%」は「剰余」つまり割った余りを求める演算なので、`x % 2 == 0` で「2 で割った余りが 0」つまり 2 の倍数という意味になります。

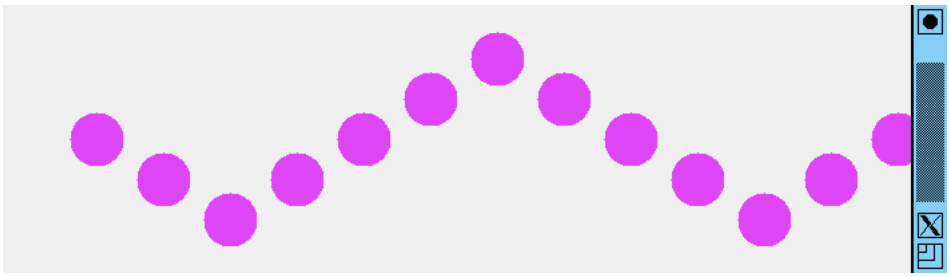
例解 3-3-a

この問題では、先の例題と同様に「動く向きを反対にする」動作を、y があ
る値より小さくなった場合にも実行します。これは、次のように 2 つ if 文
を並べてもできます。

```
if(y+80 > getHeight()) { dy = -dy; }
if(y < 50) { dy = -dy; }
```

しかし、どちらも「動作」が同じなのですから、「または」の複合条件を使っ
て 1 つの if 文にまとめた方が分かりやすく簡潔なのではないでしょうか。

```
if(y < 50 || y+80 > getHeight()) { dy = -dy; }
```

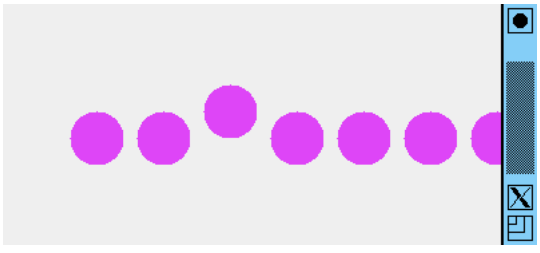


例解 3-3-b

97)0 から数えるので、3 番目というのは値 2 になります。

この問題では、一見「カウンタ i が 2 のときだけ」[97]何かすればよさそうですが、「それ以外」の場合も通常の位置に描く必要があるのですから、次のようにするのが素直でしょう。

```
public void paintComponent(Graphics g) {
  g.setColor(new Color(220, 70, 240));
  for(int i = 0; i < 20; ++i) {
    int x = 50 + i*50;
    if(i == 2) {
      g.fillOval(x, 60, 40, 40);
    } else {
      g.fillOval(x, 80, 40, 40);
    }
  }
}
```



ですが、y の値も変数に入れておくようにして、「カウンタ i が 2 のときだけ y を減らす」方法もあり得ます。

```
int x = 50 + i*50;
int y = 80;
if(i == 2) { y = y - 20; }
g.fillOval(x, y, 40, 40);
```

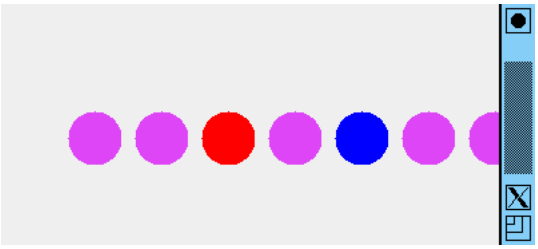
が大切だと思う人
れませんが、コン
て速いので、通
やすさと美しさ」
べきです。実際に
難かったら、その
法を検討すればよ

どちらがいいでしょう? 筆者は先に挙げた方が分かりやすくてよいと思いますが、場合によってはこちらの方がいいこともあるかも知れません。このように、プログラムを書く際には、同じ動作が何通りもの方法で書けるのが普通です。その中でどの書き方にするかは、あなたが「これが一番分かりやすくて美しい」と思う方法を自分で判断し、選ぶようにしてください。[98]

例解 3-3-c

この問題は、if-else の連鎖を用いて「赤」「青」「その他」に枝分かれするのが素直だと思います。位置は規則的なので、枝分かれの中では `setColor()` だけ実行すればよいはずです。

```
public void paintComponent(Graphics g) {
  for(int i = 0; i < 20; ++i) {
    int x = 50 + i*50;
    if(i == 2) {
      g.setColor(Color.RED);
    } else if(i == 4) {
      g.setColor(Color.BLUE);
    } else {
      g.setColor(new Color(220, 70, 240));
    }
    g.fillOval(x, 80, 40, 40);
  }
}
```



「その他の色」を毎回設定するのは無駄に感じるかも知れませんが、分かりやすさを優先すればこれでいいのではないでしょうか。[99]

99)「その他」の色を必要最小限だけ設定するように直すのはさらなる課題ということにしておきます。

13 まとめ

この章では、プログラムの中でもメソッドの内部でさまざまな動作を記述するのに必要な「繰り返し」と「枝分かれ」を学びました。これらと「上から順番に実行する」機能を併せてさまざまな実行の流れを作り出すもの**制御構造**と呼びます。制御構造はこの後もずっと使うことになるので、十分に慣れておくようにしてください。

14 付録: Java のプログラム構造

ここまで、なんとなく「見よう見まねで」Java のプログラムを書いて頂いて来ましたが、実際にはプログラムの書き方の規則は、その言語の**文法**によって定められています。

ここまでで、クラス、メソッド、制御構造について学んで、だいぶ書けるものも複雑になってきたので、ここで Java 言語の文法のまとめを示しておくことにします。[100]

100)ここで示す Java の文法は、本書の内容に合わせてやや簡略化してあります。

以下の文法規則では、「[X]」は「X があってもなくてもよい」、「X|Y」は「X または Y」、「X…」は「X を並べたもの」、「X,..」は「X をカンマで区切って並べたもの」を表すものとします。

```
プログラム ::= クラス…
クラス ::= [ 修飾…] class クラス名 [ 継承…] { 定義… }
修飾 ::= public | protected | private | static | final
継承 ::= extends クラス名 | implements インタフェース名,..
定義 ::= [ 修飾…] 変数定義 | メソッド | クラス
```

これを見ると、プログラムとはクラスが並んだもの、クラスとは「class クラス名」の後に中かっこで囲んで定義を並べたもので、修飾や継承の指定がついていることもある、などのように読んで行くことができます。

実際にプログラムを書くときはこれまで通り見よう見まねで書いていいのですが、細かいところで「ここは何が書けるの?」と疑問に思ったら、その時は文法を参照するのがよいでしょう。たとえば、定義としては変数定義、メソッドのほかにクラスも含まれるので、クラスの中にクラスを入れることもできます (これを何に使うのかはまた後で説明します)。

101)ところで、変数定義の前には「[修飾…]」がついていませんが、これは文の一種ないし一部として変数定義を書くところでは修飾がつけられないからです。このため、先の「定義」のところで変数定義の前に修飾がつけられるように指定してあります。このように、文法の定義は正確さが大切なので細かい配慮がされています。

では各種定義の内容を見てみましょう。

```
変数定義 ::= 型指定 ( 変数名 [= 式] ) ,..  ;
メソッド::= [ 修飾…] 型指定 メソッド名 ([ パラメタ,..])
                [throws 例外名,..] { 文… }
型指定 ::= 型名 | 型指定 [] | 型指定<型指定,..> | void
型名 ::= クラス名 | boolean | byte | short | char
      | int | long | float | double
パラメタ ::= 型指定 変数名
```

変数定義は、同じ型の複数の変数をまとめて定義できます。また、初期値を指定することもできます。[101]メソッドについている throws についてはまた後で説明します。型指定に [] がついているのは配列ですね。<…>がついているものについても、説明は後まわしです。

次に、具体的な動作を表す「文」について見てみましょう。

```
文 ::= 変数定義 | if(式) 文 [else 文] | while(式) 文
     | for(( 変数定義 | 式 ); 式; 式) 文
     | for(型指定 変数名:式) 文
     | { 文… } | 式; | throw 式; | break; | continue;
```

for 文の冒頭にはただの式が来る場合も、変数定義が来る場合もあります。「:」の出て来る形は **foreach ループ**と呼ばれ、配列などのすべての要素を取り出しながらループする場合に使います。[102]また、式の後ろに「;」をつけたものも文になっています。これは、代入などは式の一種だからです。最後に「式」について見てみましょう。

102)たとえば、xs が整数の配列のとき、「for(int i:xs) ...」で、変数 i に xs の各要素が順番に入って繰り返し処理がおこなえます。後の章でこのループを使ってみます。

```
式 ::= 式 演算子 式 | 演算子 式 | 式 [式]
     | new 型指定 (式,…) | new 型指定 [式]…
```

```
        | new 型指定 []{ 式,..}
        | 型指定 . メソッド名 (式,…)
        | 式 . メソッド名 (式,…) | リテラル | 変数名
        | クラス名 . 変数名 | 式 . 変数名 | ( 式 )
リテラル::= 整定数 [l] | 実定数 [f] | "文字…" | ’文字’
        | true | false | null
```

式には演算子の適用、配列参照 (添字)、メソッド呼び出し、リテラル (定数)、そして new によるオブジェクト生成があります。new の中でも配列生成の new は構文が特別で、既に学んだ要素数を指定する方法のほかに、個々の要素の初期値を指定して配列を作る方法もあることが表されています。[103)]

リテラル (定数) には論理値 (true、false)、文字列、文字、数値、そして「どのオブジェクトも参照していない」ことを表す特別な値 null があります。整数リテラルは「l」がついたのは long、そうでないのは int。実数リテラルは「f」がついたのは float、そうでないのは double のリテラルになります。

103)たとえば、「new int[]{1, 2, 3}」で、大きさが 3、各要素の値が 1 と 2 と 3 の整数配列を作り出すことができます。