6. クラスの基礎

Java は，オブジェクト指向言語であるといわれる。オブジェクト指向は，プログラミングだけでなく，そのプログラムやシステム全体の設計にも通じる考え方になってきている。ここではオブジェクト指向の考え方について解説する。その後で，Java におけるクラスの作り方や使い方の最も基本的な部分について述べる。

本章では，以下について学習する。

●オブジェクト指向プログラミングの考え方
●クラスの定義
●オブジェクトの生成とメンバへのアクセス
●メソッド

6．1　オブジェクト指向プログラミングの考え方

「オブジェクト指向」のオブジェクト（object）とは「機能的に 1 つにまためたもの」のことであり，現実の世界の槻念や具体的な物をプログラムの中で表現したものである。オブジェクト指向の考え方では，ソフトウェアを含むシステムは様々なオブジェクトが集まったものであり，オブジェクト相互のやり取りによって全体が機能するものである。例えば，自動車（別にスペースシャトルでもコンピュータでもかまわないが）のように複雑な工業製品を設計したり製作することを想像してみてほしい。複雑なものほど，適当な単位に分割して製作を進めるだろう。「適当な単位」は，一般には部品と呼ばれており，ソフトウェアの用語を使用するならオブジェクトである。エンジン，トランスミッション，ブレーキ，タイヤ，ボディなど，多くの部品が集まって 1 台の自動車が構成される。そして，エンジンのような部品もさらに小さな部品によって構成されている。ソフトウェアの構成も同じように考えることができる。

6．1．1　カプセル化

ソフトウェアにおけるオブジェクトは，データ（属性）とそれ処理するための操作（手続き）を 1 つにまとめたものである。これまで見てきた Java のプログラムでは，データは変数で，おのおのの操作はメソッドにより実現されていると理解してよいだろう。オブジェクトを操作する際は，直接オブジェクトの変数にアクセスするのではなく，あらかじめ決められているメソッド，すなわちインタフ工ース（interface）を通じて操作する。さらに言うと，インタフエースさえ分かっていれば，内部の変数を気にする必要はない。このように，オブジェクトの内部の詳細を外部から隠す（隠蔽）ことをカプセル化（encapsulation）と呼ぶ。

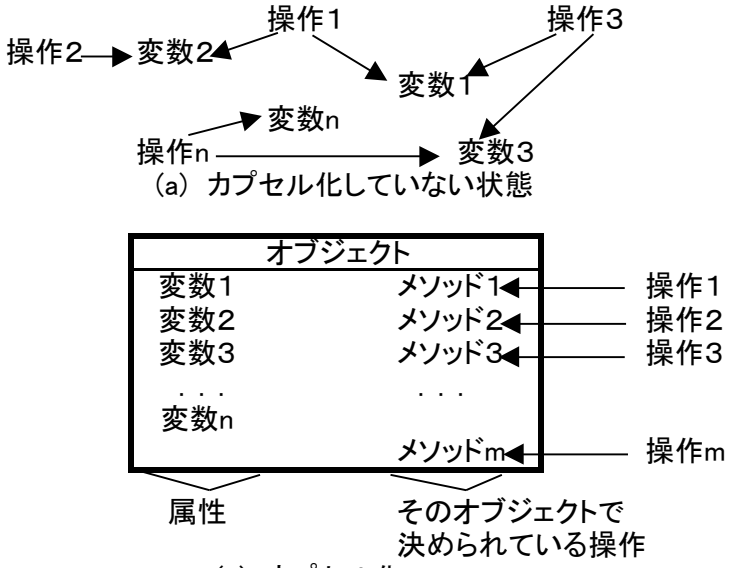

図6．1 カプセル化の概念

カプセル化の概念を図 6．1 に示す。例えば，自動車のエンジンを 1 つのオブジェクトとして考えてみよう。ソフトウェア中ではエンジンは極めて多数の変数によって表現できることは想像できるだろう。しかし，エンジンの回転数を変更するにはアクセルを操作すればよい。すなわち、エンジンの構成は隠蔽され「アクセルを操作する」ためのインタフエースで回転数を制御することができる。

詳細を隠蔽することの利点は，インタフエースに変更がない限り，オブジェクト内部の変更が外部に影響を与えないことである。オブジェクトの内部のある変数を int 型から double 型に変更しても，オブジェクトの外部には無関係である。同様にインタフエースが同じオブジェクトなら，まったく別のオブジェクトに置き換えることも可能である。

6，1．2　アクセス管理

オブジェクトに対するアクセス手段は, そのオブジェクトの作成者が定義できる。このとき、オブジェクトの外部から使用できる方法と内部でだけ使用できる方法に分けて作成することができる。すなわち、隠蔽するものとしないもの，どの範囲に対して隠蔽するかなどを指定できる。これをアクセス管理と呼ぶ。詳しくは 7．7 節で述べるが，例えば，これまでサンプルとして挙げたプログラム中の public というキーワードは、公開（どのオブジェクトからでもアクセスできる）という意味である。これとは逆に，まったく公開せずオブジェクトの内部からだけアクセスできるという指定や，オブジェクトの外でもある範囲からならアクセスできるといった指定が可能である。

6．1．3　クラスとオブジェクト

オブジェクトはクラスを元に生成される。すなわち，クラスは設計図に相当し，この設計図に従って生成されたもの（実体）がオブジェクトである。オブジェクトという用語ではなく、インスタンス（instance）という用語を使用する場合もある。そして，オブジェクト（インスタンス）を生成することをインスタンス化（instantiation）と呼ぶ。

単一の設計図からは同じ製品を複数作成することができるのと同様に，1 つのクラスについ

て複数のオブジェクトを生成することができる。たとえて言うなら，鋳型と鋳物の関係と同じである。鋳型（クラス）から，同じ形の鋳物（オブジェクト）を何個も作ることができる。
クラスとオブジェクトの関係を図 6．2 に示す。実際のプログラムの例として，OK ボタンやキャンセルボタンなどのボタンを挙げることにする。ボタンは，プログラム中のどこかでクラスとして定義されている。これを Button クラスとする。ウィンドウ上の OK ボタンは，Button クラスから生成されるオブジェクトである。同様に，キャンセルボタンも Button クラスから生成されるオブジェクトである。すなわち，Button クラスの定義はプログラム中のどこか 1 カ所にあるが，実際にボタンとして機能するのは，それから生成されたオブジェクトであり，同じクラスであっても同時に複数のオブジェクトを生成できる。

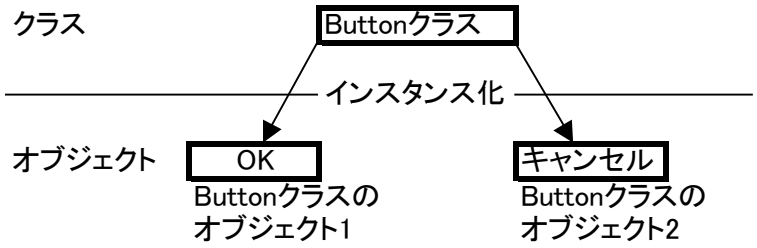

図6．2　クラスとオブジェクトの関係

6．1．4　継　承
クラスを定義する際には，クラスのすべてを記述してもよいが，元となるクラスがあればそれを拡張（extend）して定義することもできる。すなわち，似た属性を持つクラスが既にある場合に，その定義を流用して差分に相当する部分だけを改めて定義する方法である。オブジェクト指向の用語では，このような定義の方法を継承（inheritance）と呼ぶ。また，元のクラスをスーパークラス，それから新しく作られるクラスをサブクラスと呼ぶ[1]。継承については第 8 章で詳しく述べることにする。

6．2　クラスの定義
int 型や double 型は Java に組み込みの型であるが、予約語 class を使用してクラスを定義することができる。あらゆる変数に型があるように，すべてのオブジェクトは何らかのクラスに属する。オブジェクトの設計図であるクラスの定義は，クラスの宣言で記述される。クラスの宣言は以下の形式をとる。

```
修飾子　class　クラス名｛
    //　メンバの定義
}
```

ここで修飾子は public、abstruct，final、何も付けない，のいずれかである（ただし，public は他の修飾子と組み合わせることができる)。public は，既に簡単に触れたが，アクセス制御を行うための予約語のひとつで，このクラスのオブジェクトが他のオブジェクトからでもアクセスできることを示す。abstruct と final はクラスの継承と関係するので，第 8 章で説明する。

修飾子については，それぞれの説明を行うまで無視するものとし，これ以降しばらくの間，クラス宣言の修飾子は public のみを使用する。
クラス名はクラスの名前であり，変数名と同様、大文字小文字は区別される。慣例として先頭の文字を大文字とすることが多い。
クラス名の後にはブロックが続くが，このブロックの中にクラスを構成する変数やメソッドを記述する[2]。これらはメンバ（member）と呼ばれクラスを構成する変数は特にメンバ変数と呼ばれる。メソッド（またはメンバ関数）の定義については，これまで main（）メソッドについてのみ行ってきた。既に使用している System. out. println()メソッドも,main()メソッド同様に定義されていると考えていただきたい。

[2] 正確には、クラスに属するメンバと、クラスから生成されたオブジェクトに属するメンバが存在するが，ここでは区別していない。

6．2．1　メンバの定義

メンバ変数の宣言は，クラスを宣言するブロックの中で行う。メンバ変数を持つクラスの例として，プログラム 6．1 に，2 次元の座標（x，y）を表現する Point クラスの定義を示す。Point クラスでは，座標 x，y は int 型としている。Point クラスはメンバ変数だけを持ち、今のところメソッドは持っていない。

```
1:public class Point {
2: /**  座標 x, y */
3: int x, y;
4:}
```

プログラム６．１　Point クラスの宣言

演習 6．1
3 次元の座標（x，y，z）を表現するクラス Point3D を定義しなさい。座標 x，y，z はそれぞれ int 型とする。定義したらコンパイルしなさい。

メソッドの細かい書式は 6．4 節に譲るとして，ここでは，Point クラスに座標を以下のように表示する print（）メソッドを定義してみよう。

```
10、20
```

これまでのサンプルプログラムと同様，値の表示には System. out. println（）を使用する。プログラム 6．2 に，print（）メソッドを加えた Point クラスの宣言を示す。

```
1:public class Point {
2: /**  座標 x, y */
3: int x, y;
4:
5: /**  座標を表示する　*/
6: void print() {
7:   System.out.println(x + ", " + y);
8: }
```

9:}
プログラム６．２　Point クラスに対する print()メソッドの追加（Point.java）

6 行目から始まる void　print（）｛...｝が、print（）メソッドの定義である。先頭の void は、メソッドの戻り値がないことを表す。戻り値については後で述べるので、ここでは無視してよい。クラスの宣言プロックには、メンバ変数 x や y と同様に、メソッドの定義も含まれている。メソッドからは，クラス内のメンバ変数や別のメソッドにアクセスできる。この例では，print（）メソッドの中では，特別な操作なしに、メンバ変数 x、y を使用している（ここでは System. out. println（）を通じて表示する）。
ちなみに，このプログラムでは main（）メソッドを定義していないため、これまでのサンプルプログラムのように単体で実行することはできない。

演習 6．2
演習 6．1 の 3 次元の座標（x，y，z）を表現するクラス Point3D に，座標を 10、20、30 のように表示する。print（）メソッドを定義しなさい。定義したらコンパイルしなさい。

6．3　オブジェクト
クラスは、あくまでオブジェクトを生成するための設計図である。実際にクフスを使用するには、クラスをもとにオブジェクトを生成しなければならない。オブジェクトを使う為には、オフジェクトを生成（インスタンス化）してから、そのメンバ変数やメソッドにアクセスする。ここでは、オブジェクトの生成とメンバへのアクセスについて解説する。

6．3．1　オブジェクトの生成
int 型のような基本データ型では、変数を宣言するとすぐに使用することができた。

```
int　x；　　//　int 型の変数 x の宣言
x＝10＋20；　//　変数 x は基本データ型なので宣言の直後から使用できる
```

オブジェクトの場合は、変数を宣言するだけでなく、実際にはオブジェクトを生成する必要がある。
　クラスは，あたかも型のように扱うことができる。特に，クラスを型として考える場合クラス型と呼ぶことにする。例えば，Point クラスを宣言すると，その名前の Point を型の名前のように扱うことができる。また，Point クラスを型として考える場合は Point 型と呼ぶ。オブジェクトを生成するには，まず，基本データ型と同様にクラス型の変数を用意する。
　クラス型の変数は，配列と同様に参照型なので，基本型のように変数を宣言しただけでは使用できない。新しいオブジェクトを生成するには，演算子 new を使用する。演算子 new は，オブジェクトを生成して，そのオブジェクトへの参照を返す。以下のコードでは，Point 型の変数を宣言して，オブジェクトを生成し，変数に割り当てている。

```
Point　 pt；　　　　//　Point 型の変数 pt の宣言
pt ＝ new　Point（）；　// Point クラスのオブジェクトを生成
```

または，

```
Point pt ＝ new Point（）；　// Point 型の変数 pt の宣言とオブジェクトの生成
```

1 つの変数には（配列でない限り）1 つのオブジェクトが対応する。したがって，例えば Point クラスのオブジェクトを複数生成するには，変数も複数宣言して，それぞれに対してオブジェクトを演算子 new により生成すればよい。

```
Point ptl＝new Point（）；　　//　オブジェクト 1
Point pt2＝new Point（）；　　//　オブジェクト 2
```

上の例では，Point クラスのオブジェクトを 2 つ生成している。一方は変数 ptl で，他方は pt2 に割り当てられている。

プログラム 6．3 に，Point クラスのテスト用の TestPoint クラス示す。TestPoint クラスの main( )メソッドでは, Point 型の変数を宣言し, Point クラスのオブジェクトを生成している。

```
1:public class TestPoint {
2: public static void main (String argv[]){
3:   Point pt1 = new Point();
4:   Point pt2 = new Point();
5: }
6:}
```

プログラム６．３　Point クラスのインスタンス化（TestPoint.java）

6．3．2　メンバへのアクセス

オブジェクトのメンバへのアクセス、例えば Point クラスのメンバ変数 x，y や，メソッド print()アクセスする方法について述べる。オブジェクトを生成する前に，そのメンバ変数やメソッドにアクセスすることはできない。前節で述べたように，オブジェクトを生成する際に，変数にオブジェクトを割り当てる。メンバへのアクセスは、このオブジェクトを割り当てられた変数と，ドット“.”を使用して，以下のようにアクセスする。

```
Point pt＝new Point（）；
pt．x＝10；
pt．y＝20
```

上記の例では，Point クラスのオブジェクトを生成し，そのオブジェクトのメンバ変数 x と y に値を代入する。メンバ変数の場合，pt．x のように「(オブジェクトを表す）変数名．メンバ変数名」でアクセスでき，この形式で基本データ型の変数と同じように扱うことができる。

メソッドの呼び出しも同様に行うことができる。

```
pt．print（）；
```

プログラム 6．4 に，Point クラス（プログラム 6．2）のメンバにアクセスするプログラムの例を示す。このプログラムでは、Point クラスのオブジェクトを生成した後（3，4 行目)、オブジェクトのメンバ変数に直接アクセスして，値を代入し（7，8，10，11 行目)、さらに Point クラスの print（）メソッドを呼び出して，Point の値を画面に出力している（14，15 行目)。

```
1:public class TestPoint1 {
2: public static void main (String argv[]){
3:   Point pt1 = new Point();
4:   Point pt2 = new Point();
5:
6:   //メンバ変数へのアクセス
7:   pt1.x =10;
8:   pt1.y = 20;
9:
10:   pt2.x = - pt1.x;      //pt2 を pt1 と原点と対称な点とする
11:   pt2.y = - pt1.y;
12:
13:   //メソッドへのアクセス
14:   pt1.print();
15:   pt2.print();
16:}
17:}
```

プログラム６．４　メンバへのアクセス（TestPoint1.java）

プログラム 6．4 は，TestPoint クラスの main（）メソッドから別のクラスである Point クラスのメソッドにアクセスする例であることに注意をしてほしい。この場合は，本節で説明してきたように「オブジェクトを表す変数名．メンバ」という形でアクセスする。これに対して，プログラム 6．2 に示したように，Point クラスの内部で，そのクラスのメンバにアクセスする際には，特にオブジェクトを表す変数名を使用しなくてもよい。ただし，自分自身のメンバに対するアクセスであることを明示したい場合，オブジェクトを表す変数名の代わりに，そのオブジェクト自身を表す予約語 this を使用することができる。以下に，オブジェクトの内部からアクセスする場合と外部からアクセスする場合について擬似的に示す。

```
Public class MyClass｛
  int x ;
    void funcl（）｛. . .
      x ＝100 ;       // 自分のメンバに内部的にアクセスする場合
                      // は直接指定できる
      . . .
      this．x＝200 ;      //  自分のメンバに内部的にアクセスすること
                          //  を明示したい場合は予約語 this を使用する
      . . .
      Point   pt＝new Point（）;
      pt．x＝5 ;      //  他のクラスのメンバにアクセスする場合は
                      //  “変数名．メンバ”の形でアクセスする
    }
}
```

演習 6．3
演習 6．2 で作成した Point3D クラスのオブジェクトを生成し，そのメンバ変数やメソッドにアクセスするクラス（TestPoint3D）を作成しなさい。

6．4　メソッド

オブジェクトを操作する際は、そのオブジェクトが提供するメソッドを使用する。メソッドは，オブジェクトがメッセージを受け取った際に実行される，あるまとまった処理を定義したものである。歴史的には C 言語や C＋＋言語で関数と呼ばれていたものと同等のため、メンバ関数という呼び方をする場合もある。「オブジェクトにメッセージを送る」という表現が使用されるが，プログラム中ではメソッドへのアクセスすなわち呼び出しと考えることができる。メソッドからメソッドを呼び出すこともでき、プログラムは全体としてメットジの受け渡し（メッセージ・パッシング）の連続で実現される。

メソッドを呼び出す際には、引致（argument）と呼ばれる形で，値をメソッドに提供することができる。また，メソッドの実行が終了する際には，呼び出し元に値を返すことができる。これを戻り値(return　value)と呼ぶ。メソッドが呼び出されると何らかの仕事が行われるが、戻り値が得られるだけでなく，何らかの変化を引き起こす。これを副作用と呼ぶ。
自動販売機を例に考えてみる（図 6．3)。自動販売機にお金を入れて商品を指定すると商品が得られる。この例が自動販売機オブジェクトの「商品を買う」というメソッドの呼び出しだとすると、「お金」と「商品指定」は引数であり，「得られた商品」は戻り値である。また、外界からは見えないが、自動販売機の内部では売上が増加し，商品が減少するという副作用が起こる。

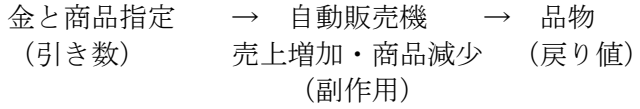

図 6．3　自動販売機の例

6．4．1　メソッドの定義

メソッドはクラス宣言のブロック内で，以下のような形式で宣言する。

メソッド修飾子　戻り値の型　メソッド名（引数リスト）{メソッドのボディ}

① メソッド修飾子
アクセス制御などのためにメソッドの性格を指定することができる。
② 戻り値の型
メソッドの実行が終了し，そのメソッドの呼び出し元に対して，出力として返す値の型を宣言する。
③ メソッド名
メソッドの名前。後で述べるが，同じ名前のメソッドを複数定義することもできる。
④ 引数リスト
メソッドの呼び出し元から値を受け取るための変数（引数）が，その型宣言とともに列挙される。
⑤ メソッドのボディ
メソッドの本体にあたる部分。

関数や引数の型は，int 型や double 型など基本データ型のほかに，配列やクラス型を書くこともできる。引き数がない場合は，引き数リストには何も書かない。戻り値の型は必ず指定する必要がある[3]。戻り値がない場合は，値がないことを示す型である予約語 void を使用する。引数がある場合は，引数リストに引数のすべてを記述する必要がある。

[3] 後で述べるコンストラクタだけは，戻り値の型の指定は不要である。

引数がない場合は，引数リストを空にする。以下に例を挙げる（ボディの中身は省略する)。

```
public class MyClass｛
  //  print（）メソッドは引き数はなく，戻り値もない公開（public）メソッド
    public void print（）{...}

  //  add（）メソッドは int 型の値 1 つを引数として，long 型の値を戻り値とする
  //  公開メソッドである。
    public long add（int x）{...}

  // distance（）メソッドは Point 型のオブジェクト，2 つを引数として，
  // double 型の値を戻り値とする公開メソッドである
    public double distance（Point ptl，Point pt2）{...}
}
```

6．4．2　メソッドの実行

Java では，メソッドを次々と呼び出すことでプログラムが実行されていく。例えば以下のように，funcl（）メソッドから func2（）メソッドを呼び出した場合について考える。

```
funcl（）{処理 a；処理 b；func2（）；処理 c；）
func2（）{処理 x；処理 y；}
```

基本的に，funcl（）では処理 a；，処理 b，func2（)，処理 c と順次実行される。前の処理が終了してはじめて次の処理が実行され，処理 a と処理 b が同時に実行されることはない。同様に，処理 b と func2（）が同時に実行されることもない。処理 b の実行が完全に終了してから func2（）が呼び出され完全に制御が func2（）に移る。処理 x，処理 y が実行されて func2（）の実行が終了すると，制御が funcl（）に戻り，処理 c が実行される（図 6．４)。

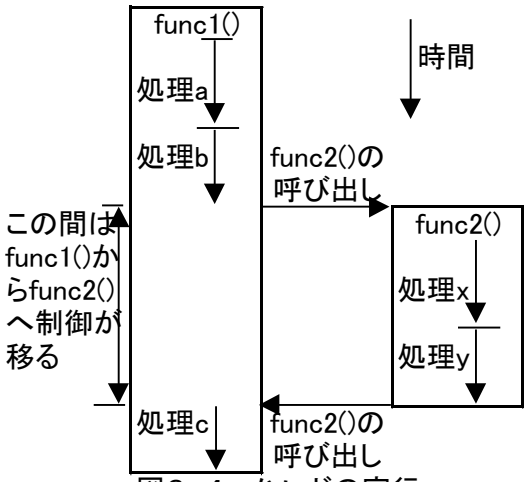

図6．4　メソッドの実行

それでは，以下のようなメソッドの呼び出しでは，実行順序はどのようになるだろうか。

```
main（）{funcl（）；}
funcl（）{... func2（）; func3（）；...}
func2（）{... func3（）；...}
func3（）{...}
```

この例では以下のように制御が移って行く。

main（）⇒funcl（）⇒func2（）⇒func3（）⇒func2（）⇒func1（）⇒func3（）⇒funcl（）

上記のメソッドの呼び出しを確かめるためのクラスをプログラム 6．5 に示す。これまではメソッドの呼び出しに着目して解説してきたが，実際のプログラムでは，オブジェクトを生成してからでないと，メソッドを呼び出しができない点に注意を要する。

このプログラムは 2 つのクラスからなっている。 TestMethodCall クラスは main（）メソッドを提供する。TestMethodCallSub クラスでは funcl（），func2（），func3（）メソッドを提供する。なお，Java では 1 つのソースファイル中に public なクラスを 1 つまで定義することができる。public でないクラスは複数定義してもかまわない。

```
1:public class TestMethodCall {
2: public static void main (String argv[]){
3:   TestMethodCallSub testObj = new TestMethodCallSub();
4:   testObj.func1();
5: }
6:}
7:
8:class TestMethodCallSub {
9: void func1() {
10:   System.out.println("func1() ... in");
11:   func2();
```

```
12:   System.out.println("func1() ... mid");
13:   func3();
14:   System.out.println("func1() ... out");
15: }
16:
17: void func2() {
18:   System.out.println("func2() ... in");
19:   func3();
20:   System.out.println("func2() ... out");
21: }
22:
23: void func3() {
24:   System.out.println("func3() ... in");
25:   System.out.println("func3() ... out");
26: }
27:}
```

プログラム６．５　メソッドの呼び出しのテスト（TestMethodCall.java）

6．4．3　値の受け渡し

制御の流れを理解したところで，メソッド間の値の受け渡し方法を見てみよう。以下の例は，クラス MyClassA の funcA（）メソッドからクラス MyClassB の funcB（）メソッドを呼び出す例を示している。FuncB（）メソッドは char 型と double 型の値を受け取り、int 型の値を返すメソッドであるとする。

```
class   MyClassA {
     void funcA（）{
          int c；char a；double b；
          MyClassB obj＝new MyClassB（）；
          c＝obj．funcB（a，b）；...
  }
}
class MyClassB {
  int funcB（char x、double y）{
          int z；    ...
          return z；
  }
}
```

この例では，クラス MyClassA の funcA（）でクラス MyClassB のオブジェクトを生成し，そのメソッドである funcB（）を呼び出している。ここで，funcA（）の変数 a，b，c，obj は，funcA（）のブロック内でだけ有効な変数である。このような変数をローカル変数と呼ぶ。あるメソッドのローカル変数の値を他のメソッドで直接変更することはできない。そこで，メソッドを呼び出す際の引数とメソッドから戻る場合の戻り値を通して，呼び出す側のメソッドと呼び出される側のメソッドの間で値をやり取りすることができる。図 6．5 に，引数と戻り値の概念を示す。

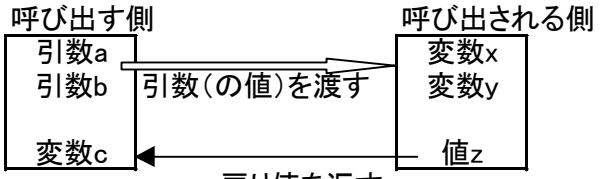

図6．５　引数と戻り値の概念

メソッドが呼び出される際には，メソッドの宣言の引数が呼び出し側で指定された引数の値で初期化されてから制御が移動する[4]。上記の例では，funcB（）を呼び出す側の引数は a と b が指定されているので，funcB（）に制御が移動する前に，x と y は a と b の値で初期化される。x と y は，funcB（）のブロック内ではローカル変数として扱われる。したがって，呼び出されたメソッドの中で値を変更しても呼び出した側には影響しない。メソッドを呼び出す際に引数として指定する値の方は，呼び出されるメソッドの宣言で指定された引数の型に一致するか，そうでなくとも，自動的な型変換によって一致する必要がある。上記の x や y が，引数として渡された a や b で初期化されることを考えれば理解できるだろう。

実際にプログラムで引数を使ってみよう。Point クラス（プログラム 6．2）に座標を設定する set（）メソッド[5]を追加した例を，プログラム 6．6 に示す。他のメソッドは省略した形で以下に示す。

```
1:public class Point1 {
2: /**  座標 x, y */
3: int x, y;
4:
 /**  座標を表示する　*/
 void print() {
   System.out.println(x + ", " + y);
 }
6:
7: /**  座標を設定する　*/
8: void set(int ptx, int pty) {
9:    x = ptx;
10:    y = pty;
11: }
12:}
```
プログラム６．６　引数を持つ set()メソッド（Point.java）

演習 6．4
演習 6・2 で作成した Point3D クラスに，座標を設定する set（）メソッドを定義しなさい。

あるメソッドからそのメソッドを呼び出した側に制御を移動させる場合は，return 文を使用する。return 文は以下の形式をとる。

return 式；または return；

メソッドの最後や途中で return 文が実行されると，その場所でメソッドの実行を終了（中断）

し，呼び出した側に制御が移る。メソッドの宣言で戻り値の型が宣言されている場合には，return 文の後に式が必要である。その式の値が，呼び出した側でのメソッドの値（戻り値）として渡される。プログラム中でメソッド funcB（）の戻り値を変数 c に代入したい場合には，以下のように記述する。

```
c＝obj．funcB（a，b）；　//　funcB（）の戻り値は変数 c に代入される
```

もしも，呼び出し元で戻り値を無視したい場合は，以下のように，単純にメソッドを呼び出すだけでよい。

```
obj．funcB（a，b）；　　// funcB（）に戻り値がある場合は無視される
```

実際にプログラムで戻り値を使ってみよう。Point クラス（プログラム 6．6）に，x 座標を取得する getx（）メソッドと y 座標を取得する gety（）メソッド[6]を追加した例をプログラム 6．7 に示す。他のメソッドは省略した形で以下に示す。また，この Point クラスの set（），getx（），gety（），それぞれのメソッドの使用例をプログラム 6．8 に示す。

```
1:public class Point1 {
2: /**  座標 x, y */
3: int x, y;

 /**  座標を表示する　*/
 void print() {
   System.out.println(x + ", " + y);
 }
6:

 /**  座標を設定する　*/
 void set(int ptx, int pty) {
   x = ptx;
   y = pty;
 }

7: /** x 座標の取得　*/
8: int getx() {
9:　 return x;
10: }
11:
12: /** y 座標の取得　*/
13: int gety() {
14:　 return y;
15: }
16:}
```

プログラム６．７　戻り値を持つメソッド getx()と gety()（Point1.java）

[6] 一般に、オブジェクトのインスタンス変数の値を取得するメソッドを Accessor　method と呼

び，慣例としてメソッド名を get で始める場合が多い。

```
1:public class TestPoint2 {
2: public static void main (String argv[]){
3:   Point1 pt = new Point1();
4:   pt.set(10,20);            //set()メソッドの使用例
5:
6:   int x = pt.getx();     //getx()メソッドの使用例
7:   int y = pt.gety();     //gety()メソッドの使用例
8:
9:   pt.print();
10:   System.out.println(x + ", " + y);
11: }
12:}
```

プログラム６．８　Point クラスのメソッドの使用例（TestPoint2.java）

演習 6．5
演習 6．2 で作成した Point3D クラスに，各座標を取得するメソッドと，与えられた座標との距離を計算する distance（）メソッドを定義しなさい。

7. クラスとオブジェクトの操作

本章では，クラスとオブジェクトの操作を中心として，より重要な概念について学習していく。オブジェクト指向プログラミング言語では一般的な話題も多いが，初学者は難しく感じる内容も多いかもしれない。オブジェクト指向言語では，メソッドひとつとってみても，非常に柔軟で強力な機構を備えている。例えば，クラス内で同じ名前のメソッドを複数定義できたり，オブジェクトを生成したときに自動的に呼び出されるメソッドを定義できる。これまで，クラスは設計図でオブジェクトは個々の製品であるという例のように，クラスを概念的なものとして扱ってきた。しかし，実際には，オブジェクトに属するメンバが存在するように，クラスに属するメンバも存在する。

本草では，以下について学習する。

●メソッドのオーバーロード
●コンストラクタ
●オブジェクトの参照と関係演算
●オブジェクトのメンバとクラスのメンバ
●アクセス制限

7．1　メソッドのオーバーロード

メソッドを呼び出す場合，メソッド名から呼び出すメソッドが決定されているように思うかもしれない。実際には，メソッドはメソッド名，属するクラス，引数の個数と型で識別され　メソッド名だけで区別されているわけではない。

異なるクラスに同じメソッド名のメソッドが存在しても，これらのメソッドは互いに別のものとして扱われる。これは直感的に理解できるであろう。例えば，これまでサンプルプログラムと演習で扱ってきた Point クラスの print（）メソッドと Point3D クラスの print（）メソッドは同じメソッド名を持つが，きちんと区別されて呼び出される。Point クラスでも Point3D クラスでもない，新しいクラスに print（）メソッドが存在しても，同じ理由により問題とはならない。

同じクラス内で同じメソッド名のメソッドが存在しても，引数の型や個数から区別できれば互いに別のメソッドとして扱われる。特に，クラス内で同じメソッド名のメソッドを複数作ることをメソッドのオーバーロード（多重定義）と呼ぶ。

実際にプログラムでメソッドをオーバーロードしてみよう。Point クラスに，指定された点との距離を求める distance（）メソッドを追加した例をプログラム 7．1 に示す。点の座標の指定は、x 座用と y 座標を 2 つの引数として指定する場合と Point クラスのオブジェクトで指定する場合が考えられる。

```
1:public class Point2 {
2: /**  座標 x, y */
3: int x, y;

 /**  座標を表示する　*/
 void print() {
   System.out.println(x + ", " + y);
 }

 /**  座標を設定する　*/
```

```
 void set(int ptx, int pty) {
   x = ptx;
   y = pty;
 }

 /** x 座標の取得　*/
 int getx() {
   return x;
 }

 /** y 座標の取得　*/
 int gety() {
   return y;
 }

7: /**  座標の点との距離を求めるメソッド１　*/
8: double distance(int ptx, int pty) {
9:    double d;
10:   int dx = ptx - x;                  //x 座標の差
11:   int dy = pty - y;                  //y 座標の差
12:   d = Math.sqrt(dx * dx + dy * dy);            //距離を求める
13:   return d;
14: }
15:
16: /**  指定の点との距離を求めるメソッド２　*/
17: double distance(Point2 pt) {
18:   double d;
19:   int dx = pt.x - x;                //x 座標の差
20:   int dy = pt.y - y;                //y 座標の差
21:   d = Math.sqrt(dx * dx + dy * dy);               //距離を求める
22:   return d;
23: }
24:}
```

プログラム７．１　メソッドのオーバーロード（Point2.java）

この例では，2 つの distance（）メソッドが定義されている（すなわち，オーバーロードされている)。各メソッドでの処理はほとんど同じで，自分自身の座標（メンバ変数の x と y）と引数で与えられた座標について，x 座標の差 dx と y 座標の差 dy を計算し（10，11 行目あるいは 19，20 行目)、dx と dy の二乗の和の平方根を計算し（12，21 行目)，その結果を戻り値として返す。この例で，Math．Sqrt（）メソッドは，引数の平方根を計算して，戻り値として返すメソッドである。
2 つの distance（）メソッドは、引数の個数と型により区別される。すなわち，distance（）メソッドを呼び出す際に、引数が 2 つの int 型の値であった場合はメソッド 1 が呼び出され，引数が 1 つで Point 型である場合はメソッド 2 が呼び出される。

```
Point2 ptl，pt2 ;
```

```
. . .
d＝ptl. distance（0，0）；　//　メソッド 1 の呼び出し
d＝ptl. distance（pt2）；　　//　メソッド 2 の呼び出し
```

メソッドのオーバーロードとは関係ないが，プログラム 7．1 の distance（Point　pt）メソッドの定義を見て、Point クラスのメソッドの引数が Point 型であることから，混乱する読者がいるかもしれない。 Point クラスの定義中に（定義が終わっていないのに），そのクラスのメソッドの中で Point 型の変数を使用していることになるからである。Java では，あるクラスの定義の中で，そのクラス自身を使用してもよく，このような定義は問題ない。したがって，例えば Point クラスの中に main（）メソッドを定義し，その中で Point クラスのオブジェクトを生成するようなプログラムを書くこともできる。このことを利用して，プログラム 7．2 では，Point クラスに Point クラス自身をテストする main（）メソッドが追加されている。

```
1:public class Point2 {
2: /**  座標 x, y */
3: int x, y;

 /**  座標を表示する　*/
 void print() {
   System.out.println(x + ", " + y);
 }

 /**  座標を設定する　*/
 void set(int ptx, int pty) {
   x = ptx;
   y = pty;
 }

 /** x 座標の取得　*/
 int getx() {
   return x;
 }

 /** y 座標の取得　*/
 int gety() {
   return y;
 }

 /**  座標の点との距離を求めるメソッド 1　*/
 double distance(int ptx, int pty) {
   double d;
   int dx = ptx - x;                  //x 座標の差
   int dy = pty - y;                  //y 座標の差
   d = Math.sqrt(dx * dx + dy * dy);            //距離を求める
   return d;
 }
```

```
 /** 指定の点との距離を求めるメソッド２　*/
 double distance(Point2 pt) {
  double d;
  int dx = pt.x - x;              //x 座標の差
  int dy = pt.y - y;              //y 座標の差
  d = Math.sqrt(dx * dx + dy * dy);              //距離を求める
  return d;
 }
6:
7: public static void main(String argv[]) {
8:   Point2 pt = new Point2();
9:   pt.set(10,20);
10:   //distance()メソッド１のテスト
11:   double dist = pt.distance(0,0);          //原点からの距離を求める
12:   System.out.println("Dist (0,0) - (10,20): " + dist);
13: }
14:}
```

プログラム７．２　クラスの中でそのクラスのオブジェクトを生成する例

演習 7．1
int 型の引数を 1 つだけとる test（）メソッドを定義し，引数の型を変えてオーバーロードしなさい（引数の型が long 型や short 型の test（）メソッドを同じクラスに定義しなさい)。実際にいろいろな型の引数を渡して，どの test（）メソッドが呼ばれているか調べなさい。

7．2　コンストラクタ

コンストラクタ（constructor）は，オブジェクトが生成されたときに自動的に呼び出されるメソッドである。これまでオブジェクトの初期化については特に述べてこなかったが，このコンストラクタを利用してオブジェクトを初期化することができる[1]。
コンストラクタは，「クラス名と同じ名前で，戻り値のないメソッド」である。

プログラム 7. 3 に, Point クラスにコンストラクタの定義を追加した例を示す。この例では，はじめに簡単なコンストラクタの例として引数のないコンストラクタ 1 を定義している（6 行目)。このコンストラクタ 1 では，オブジェクトが生成される際に，明示的に座標を（0，0）に設定している。

コンストラクタは, 引数を持つことができる。また，コンストラクタをオーバーロードする，すなわち，複数のコンストラクタを定義することもできる。この例では，コンストラクタ 1 に加えて，引数で座標を指定するコンストラクタ 2 を定義している。

[1]　C＋＋言語などでは，オブジェクトを消去する際に自動的に呼び出されるデストラクタ（destructor）が定義できるが，Java ではデストラクタはない，近い機能としてフアイナライザがあるが，デストラクタとは異なる。

```
1:public class Point {
2: /**  座標 x, y */
3: int x, y;
```

```
4:
5: /**  コンストラクタ１　引数なし　*/
6: Point() { x=y=0;}
7:
8: /**  コンストラクタ２　初期値となる座標を指定　*/
9: Point(int ptx,int pty){
10:     x=ptx; y=pty;
11: }
12: ...
13:}
```

プログラム７．３　コンストラクタ

実は，オブジェクトを生成する際の演算子 new の後に指定するのは，コンストラクタであると考えてよい[2]。コンストラクタは，演算子 new との組み合わせでのみ使用できる。

以下に，プログラム 7．3 で定義したコンストラクタを使用して Point クラスのオブジェクトを生成する例を挙げる。

```
//  オブジェクトを生成し，初期化する
   Point ptl＝new Point（）；              //  コンストラクタ 1 が呼び出される
                                          //  （0，0）で初期化される
   Point Ptl＝new Point（10，20）；   //  コンストラクタ 2 が呼び出される
                                          //（10，20）で初期化される
```

コンストラクタは，他のメソッドとは異なり，通常メソッドと同じ呼び出し方（演算子 new を使用しない）をしてはならない。

[2]　コンストラクタを定義していない場合，Java が自動的に生成するデフォルトのコンストラクタが呼び出される。

```
   Point pt＝new Point（）；
   Pt．Point（30，20）；          //  誤り！新しい座標で再初期化のつもりかもしれないが
                                  //  コンストラクタをこのように使用することは不可
```

コンストラクタから他のメソッドを呼び出すことができる。既にプログラム 6．6 で点の座標を設定する set（）メソッドを定義しているが，プログラム 7．3 のコンストラクタ 2　Point（int　x，int　y）は，この set（）メソッドを呼び出して以下のようにも定義できる。

```
   public class Point｛
      //  中略...
      /＊＊コンストラクタ 2 初期値となる座標を指定＊/
      Point（int ptx，int pty）｛
        //  x＝ptx；　y＝pty；の代わりに set（）を呼び出す
        set（ptx，pty）；
    }
    /＊＊座標を設定する＊/
```

```
    void set（int ptx，int pty）{
        x ＝ptx ；   y＝pty ；
    }
    ...
}
```

コンストラクタから別のコンストラクタを呼び出すこともできる。これは，「コンストラクタは new との組み合わせでのみ使用できる」というルールの例外で，呼び出し方も一般的なメソッドの呼び出しとは異なる。このような場合は，コンストラクタの名前の代わりに予約語 this を用いて，そのクラスのコンストラクタを表す。例えば，コンストラクタ 1 の中でコンストラクタ 2 を呼び出すには，以下のように予約語 this を使用する。

```
...
/＊＊コンストラクタ 1 引数なし＊/
Point（）{
    this（0，0）；  //  コンストラクタ 2  Point（x，y）の呼び出し
}
```

7．3　オブジェクトの参照と関係演算

オブジェクトは，参照型の変数を通じてアクセスする。参照型の変数には，オブジェクトそのものが保持されるのではなく，オブジェクトへの参照が保持される。このような状態を，「変数 x はオブジェクト a を参照する」などと表現する。以下に，参照型の変数で注意すべきことについて述べる。

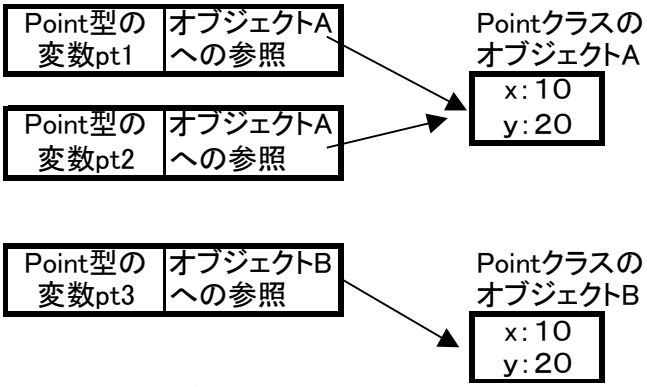

図 7．1　オブジェクトの参照の例

参照型の変数の場合，視数の変数が同じオブジェクトを参照することもある。以下は，Point 型の変数 ptl，pt2，pt3 に対して，いずれも座標（10，20）を格納する Point 型のオブジェクトの参照を代入する例である。

```
Point ptl＝new Point（10，20）；  //  ここで生成されたオブジェクトを A とする
Point pt2＝ptl ;
Point pt3＝new Point（10，20）；  //  ここで生成されたオブジェクトを B とする
```

この例が実行された結果を図 7．1 に示す。変数 ptl は，新たに生成された Point クラスのオブジェクト A を参照する。変数 pt2 に対しては，新しいオブジェクトを生成せずに，変数 ptl と同じオブジェクトへの参照が代入される。すなわち，ptl と pt2 はともにオブジェクト A を参照することになる。変数 pt3 には，変数 ptl や pt2 とは異なるオブジェクト B の参照を代入する。ただし，このオブジェクト B もオブジェクト A と同じ座標（10，20）を格納している。

ここで，変数 ptl と pt2 は同じオブジェクト A への参照を保持しているので，そのどちらからでもオブジェクト A の値を変更することができる。

```
ptl．set（20，30）；   //  オブジェクト A の値を（20．30）に変更
pt2．print（）；       //  変数 pt2 も同じオブジェクト A を参照しているので
                      //  “20，30”と表示される。
```

参照型の変数に対する関係演算は，直感と異なる点があるので注意する必要がある。参照型の変数に対する関係演算では，厳密にはその変数が保持している参照について演算が行われる。例えば，もう一度図 7，1 の状態において，以下のようなプログラムを実行するとどのようになるだろうか。

```
if（ptl＝＝pt3）{
    x ＝  true ;
} else {
    x＝false ;
}
```

この例の参照型に対する関係演算“ptl＝＝pt3”は，おのおのの変数が参照しているオブジェクトが同じかを判定していることになり，おのおのの変数が参照しているオブジェクトの内容を比較しているわけではない。したがって，上記の変数 x は false になる。

オブジェクトの間でそれぞれの内容の関係（同じか否か，大きいか，小さいかなど）を調べるには，それぞれのメンバ変数の関係を直接調べる必要がある。そうでなければ，関係を判定するメソッドを定義し，それを使用する。

例えば，Java で文字列を扱う際に使用される String クラスでは，文字列の比較には equals（）メソッドを使用する。equals（）メソッドは、引数に与えられたオブジェクトと自分の保持している文字列を比較して，一致する場合は true を，一致しない場合は false を返す。以下に，equals（）メソッドの使用例を示す。

```
String  strl，str2 ;
...
if（strl  equals（str2））{  //  strl の内容と str2 の内容が等しければ
...
}
```

7．4　ガーベージコレクションとフアイナライザ

オブジェクトを生成する際，そのオブジェクトを格納するためのメモリが確保される。「オブジェクトの生成」の反対にあたる操作は「オブジェクトの破棄」である。オブジェクトが不要となったときは，そのメモリも開放する必要がある。実際，C＋＋言語などでは，プログラム

中で明示的にオブジェクトを破棄しなければならないことがある。
Java では，プログラム中でオブジェクトを破棄する必要はなく，システムが不要なオブジェクトを判断して，自動的にメモリの開放を行う。これをガーベージ・コレクション（garbage collection）と呼ぶ。どの変数からも参照されていないオブジェクトがガーベージ・コレクションの対象となる。

実際には，オブジェクトを参照している変数が 1 つもない状態になったときに，そのオブジェクトを破棄できると判断する。例えば，あるローカル変数に対してオブジェクトを生成して，そのメソッドの中でだけ使用した場合，メソッドの実行が終了するとローカル変数は消滅する。この時点ではオブジェクトは存在しているが，どの変数からも参照されていない状態になる。オブジェクトを参照する変数に null や他の参照を代入した場合も同じである。

オブジェクトが破棄されるときに，そのオブジェクトの finalize（）メソッドが自動的に呼び出される。このメソッドをフアイナライザ（finalizer）と呼ぶ。実際にオブジェクトが削除される前に何らかの後処理が必要な場合は，finalize（）メソッドを定義しておけばよい。

ただし，フアイナライザについては，呼び出されるタイミングが特定できないことに注意する必要がある。また，プログラムが終了した場合，その時点で存在していたオブジェクトに対するフアイナライザの呼び出しが省略されることもある。

7．5　オブジェクトのメンバとクラスのメンバ

Java では、オブジェクトごとのメンバに加えて，クラス単位で存在するメンバも定義できる。

これまで扱ってきたオブジェクトのメンバ変数は，オブジェクト単位で存在している。例えば，2 つの Point 型のオブジェクト ptl と pt2 について考えると，ptl には ptl のメンバ変数 x と y が，pt2 には pt2 のメンバ変数 x と y がそれぞれ独立に存在している。このように，オブジェクトごとに存在する変数を，メンバ変数の中でもインスタンス変数と呼ぶ。

これに対して，クラス単位で存在するメンバ変数がある。このメンバ変数はクラス変数と呼ばれ、そのクラスに属するすべてのオブジェクトで共有される。すなわち，そのクラスのオブジェクトから等しくアクセスできるが，実体はクラスに 1 つである。メソッドについても同様に，クラス単位で定義されたものが存在し，クラス単位のものをクラスメソッド，オブジェクト単位のものをインスタンスメソッドと呼ぶ。

7．5．1　static なメンバ

クラス変数やクラスメソッドは，その宣言に予約語 static を付ける。これまでのサンプルの main（）メソッドは，static なメソッドの例である。以下に，クラス変数の宣言の例を挙げる。

```
public class MyClass  {
    static int a ;    // クラス変数
    int b ;           // インスタンス変数
    . . .
```

オブジェクトのメンバには，オブジェクトが生成された後でないとアクセスできない。これに対して，クラスのメンバに対しては，オブジェクトが存在していなくても[3]アクセスできる。上記の MyClass クラスのクラス変数にアクセスするには，オブジェクトを表す変数名の代わりに，クラス名を用いて以下のようにする。

```
MyClass．a＝10 ;   //  MyClass クラスのクラス変数 a に値 10 を代入
```

[3] クラスが存在すれば、すなわち，クラスがロードされていればの意。通常は特に気にする必要はないだろう。

プログラム 7．4 に，static な変数とメソッドの使用例を示す。この例に示される TestStatic クラスは、クラス変数として a を，インスタンス変数として b を定義している（2，3 行目）。メソッドについても，それぞれの変数に値を代入する seta（）（static メソッド）と setb（）がある。これまでのサンプルプログラムでは，あるクラスをテストするために別のクラスで main（）メソッドを定義してきた。しかし，本プログラムのように，自分自身のクラスのオブジェクトを生成する main（）メソッドを定義することができる（12 行目）。main（）メソッドは static メソッドであり，そのクラスのオブジェクトの存在とは無関係に実行できることが分かれば，違和感は少ないだろう。
プログラム 7．4 の main（）メソッドで、TestStatic クラスのオブジェクトを生成する前にクラス変数である x に static メソッド seta（）を用いて値を設定し（13 行目），その値を表示をしている（14 行目）。その後，2 つのインスタンスを生成し，今度は片側のオブジェクト（obj1）で x の値を変更し（24 行目），他方のオブジェクト（obj2）からも表示している。

```
1:public class TestStatic {
2:   static int a;     //クラス変数
3:   int b;               //インスタンス変数
4:
5:   /** static メソッド */
6:   static void seta(int x) { a = x; }
7:
8:   /** 通常のメソッド */
9:   void setb(int x) { b = x; }
10:
11:   /** 自分自身(TestStatic)クラスをテストする main()メソッド */
12:   public static void main(String argv[]) {
13:      TestStatic.seta(10);
14:      System.out.println("Before the instantiation: " + TestStatic.a);
15:
16:      TestStatic obj1 = new TestStatic();
17:      TestStatic obj2 = new TestStatic();
18:      System.out.println("Two Objects are created. ");
19:      System.out.println("TestStatic.a = " + TestStatic.a);
20:      System.out.println("obj1.a = " + obj1.a);
21:      System.out.println("obj2.a = " + obj2.a);
22:
23:      obj1.seta(20);
24:      System.out.println("obj1.seta(20);");
25:      System.out.println("TestStatic.a = " + TestStatic.a);
26:      System.out.println("obj1.a = " + obj1.a);
27:      System.out.println("obj2.a = " + obj2.a);
28:   }
29:}
```

プログラム７．４　static な変数とメソッドの例（TestStatic.java）

プログラム 7．4 の実行結果を以下に示す。クラス変数がオブジェクト間で共有されていることが分かる。

```
Before the　instantiation：10
Two Objects are created.
TestStatic．a　＝　10
objl．a＝10
obj2．a＝10
objl．seta（20）；
TestStatic．a　＝20
objl．a＝20
obj2．a＝20
```

7．5．2　クラス変数の初期化と static イニシヤライザ

オブジェクトが生成されるときには，コンストラクタでメンバ変数の初期化を行うことができた。クラス変数が使用されるのは，オブジェクトが生成される前かもしれないし，そもそもオブジェクトを生成しないクラスかもしれない。このため、クラス変数に対しては，コンストラクタとは別の初期化方法が必要である。

最も簡単な方法は，クラスの定義で初期値を指定しておく方法である。以下のようにすると，クラス変数 x は 10 に初期化される。実は，メンバ変数に対しても同じように初期化することができる。

```
public class TestStatic｛
  static int　x＝10；　　//　クラス変数 x を 10 で初期化
  int y＝20；
  ...
｝
```

配列やオブジェクトをクラス変数として持つなど，さらに複雑な初期化を行いたい場合，static イニシヤライザを使用する。static イニシヤライザの形式は，コンストラクタとは異なり、クラス名とは無関係で，クラス宣言の中で以下の形式で処理を記述する。

```
static ｛初期化処理｝
```

以下の例では，クラス変数として配列をセットアップしている。配列やクラス型は参照型であったことを思い出そう。したがって，単に宣言（この場合はクラス変数として宣言）するだけではなく，演算子 new を使用してオブジェクトを生成する必要がある。この例の static イニシヤライザでは、int 型 10 個分の配列を生成し，それぞれの要素の値が要素番号の二乗になるように初期化している。

```
public class TestStaticArray　｛
        static int［］array；
        static｛
            array＝new int [10]；
```

```
            for（int i＝0 ； i＜array. 1ength ； i++）
                array［i］＝i＊j ;
        }
        ...
    }
```

なお, Java では、配列名. 1ength で配列の長さを知ることができる。この例で“array. 1ength”’は配列 array の長さ 10 を表す。

7．6　final 変数

メンバ変数の宣言に予約語 final を付けることによって，その変数の値が変更の対象にならないことを表すことができる。特によく利用されるのは，static と組み合わせて実質的に定数を宣言する使用方法である。

以下は，定数として円周率（PI）を宣言している。final 変数をいったん初期化した後は，その値を変更することはできない。

```
final static double Pl＝ 3. 14159265 ;
```

7．7　アクセス制御

ここまで，アクセス制御にはほとんど触れずに話を進めてきた。アクセス制御により，クラス内のすべての変数やメソッドに対する外部からのアクセスを制御することが可能である。いくらクラスを作成しても, そのメンバ変数の値を誰でも自由に変更できたのでは, 意味がない。また, 2 つのメンバ変数で, ある一定の関係が保たれるように値を変更したいといった場合も，一方のメンバ変数だけを変更できてしまうようでは，クラスが正常に機能することを保証できない。したがって，クラスを作成し，アクセス制限によりある決められたインタフェースを利用してオブジェクトを操作するようにすることは，オブジェクト指向にとって非常に重要である。

アクセス制御には，以下のような予約語を使用する。

public	すべてのクラスからアクセス可
protected	同じパッケージのクラスとサブクラスからアクセス可
＜アクセス制限指定なし＞	同じパッケージのクラスからアクセス可
private	そのクラスの内部からだけアクセス可

public は, そのメンバに対するアクセスの制限がないという意味になる。したがって, public なメンバ変数はあらゆるクラスのメソッド内で自由に値を変更できるし，public なメソッドならどんなクラスからでも呼び出すことができる。これに対して private は，外向きには一切公開せず，クラスの内部でだけ使用するという意味になる。private なメンバに外部のクラスからアクセスするようなプログラムはコンパイルエラーを起こす。

残りの protected と「アクセス制限指定なし」は，部分的な公開であると考えてよいだろう。Java では，互いに関連のあるクラスを集めてパッケージ（package）と呼ばれる単位を定義することができる。特にアクセス制限が指定されていないメンバについては、同じパッケージに属するクラスのオブジェクトからだけアクセスすることができる。

protected は，継承と関係のあるアクセス制限であ。継承についてはまだ学習していないが，

あるクラスを元にして，別のクラスを新たに定義することができる。protected が指定されたメンバには，同じパッケージのクラスのオブジェクトに加えて、継承により作成されたクラス（サブクラス）のオブジェクトからもアクセスすることができる。

8. 継　承

オブジェクト指向プログラミングの大きな概念として，クラスの継承がある。継承を利用すると，既に存在するクラスを拡張した新しいクラスを作成することができる。継承は、単なるプログラミングの手法であるだけではなく，システムを考える上で重要なヒントとなることがある。ここでは継承とその意味について説明し，実際に Java で継承を使用する方法について解説する。

本章では，以下について学習する。

●継承とその意味
●継承と型
●スーパークラスのメンバへのアクセス
●抽象クラスと final クラス
●インタフエース

8. 1　継承とその意味

新しいクラスを定義するときに、非常に似たクラスが既に定義されている場合がある。このようなとき，定義済みのクラスはそのまま手を入れずに，しかも，そのクラスを拡張して新しいクラスを定義することができる。これを継承と呼ぶ。継承では，既存のクラスからメンバの定義を受け継ぐことができる。受け継いだメンバは，新しいクラスで定義したメンバと同じように使用できる。新しいクラスでは，変更点だけを追加することで必要な機能を実現する。継承において，元となるクラスをスーパークラス、継承して新たに作成されるクラスをサブクラスと呼ぶ。あるスーパークラス A から継承してサブクラス B を定義することを，「A を派生して B を作る」のようにも表現する。継承を利用すると，先に抽象的なクラスを定義して，後から具体的なクラスを定義することができる。

継承については，動物の例がしばしば紹介される。犬クラス，猫クラスなど，具体的な動物に対応するクラスはすぐに想像できるだろう。しかし，継承を効果的に使用するなら，動物クラスという抽象的なクラスを定義する方法が考えられる。動物クラスには「食べる」,「移動する」,「鳴く」など，どの動物にも共通する特徴や動作を定義し，具体的な動物に対応するクラスは動物クラスを継承して作成する。

実際のプログラムに即した例として，ウィンドウやダイアログ中に表示されるボタン（OK／キャンセルボタンなど）を考えてみる。どんなボタンでも，画面上に「表示する」こと，マウスクリックで「クリックする」こと，そのときにボタンに対応付けられた「動作を引き起こす」ことができる。そこで，最も基本的なクラスとして Button クラスを定義し，そこから様々なバリエーションを派生することが考えられる。ボタンの中に，「OK」などの文字の代わりに絵を表示することのできる PictureButton クラスや，ボタンを押したときに 3 次元的なアニメーションを行う Button3D クラスなどである。このようなボタンはいずれも，「表示する」,「クリックする」,「動作を引き起こす」という基本的な部分については Button クラスと共通なので，Button クラスを継承して定義することができる。

動物とボタンにおける例で重要なのは，概念上，サブクラスはスーパークラスの一種であるということである。犬は動物の一種（犬クラスは動物クラスの一種）だし，3 次元表示ボタンは単純なボタンの一種（Button3D クラスは Button クラスの一種）である。このような関係を is－a 関係と呼ぶ。プログラムを作成する際に，継承により作成が可能な場合でも，概念的に is－a 関係が成立しているかは考慮すべきである。例えば，色は赤（R），緑（G），青（B）の合成として表すことができ，それぞれの成分を int 型であるとすると，以下のように定義することができる。

/＊RGＢによる色を表現する Color クラス＊/
class Color　｛　int r ;　int g ;　int b｝

前章で使用した 2 次元の座標を表す Point クラスは，x 座標と y 座標の 2 つの int 型のメンバ変数からなるクラスであったことを思い出そう。

/＊　2 次元の座標を表す point クラス
class Point　｛　int　x ; int y ; ｝

ここで，Point クラスを継承して、Color クラスを作成することは正しいだろうか。プログラムの上では, Point クラスを継承して int 型のメンバ変数を 1 つ加えれば Color クラスと同等の構造になる。しかし、直線や図形を Point クラスのオブジェクトで定義できても，Color クラスのオブジェクトで定義することはできない。言うまでもなく，Color オブジェクトは Point オブジェクトの一種ではなく，is－a 関係が成立するか考えると問題があることが分かる。このように，概念上問題がある継承は原則として避けるべきである。
さて，今度は図形を表すクラス、例えば三角形を表す Triangle クラスについて考えてみる。三角形は 3 つの点によって定義できるので、3 つの Point クラスのオブジェクトをメンバに持つ。また，線や内部領域を描画するときの色として，Color クラスのオブジェクトもいくつかメンバに加える必要があるかもしれない。このように, Triangle クラスは Point クラスや Color クラスのオブジェクトをメンバとして持つことが分かる。このような関係を has－a 関係と呼ぶ。

8．2　継承によるクラスの作成
継承を利用して新しいクラスを定義するには、クラスの宣言時にどのクラスを継承するのかも併せて記述する。クラス宣言で予約語 extends を使用して，以下のような形式で宣言する。

class サブクラス名 extends スーパークラス名｛...｝

extends に続いてスーパークラスを 1 つだけ指定する。これは，Java では直接，複数のクラスを継承することはできないことを意味する。ここで指定するスーパークラスが何かのサブクラスであってもかまわない。すなわち、サブクラスをさらに継承して新しいサブクラス（もともとのスーパークラスから見ると孫にあたるクラス）を作成することができる。
既存のクラスを継承して新しいクラスを作成した場合でも、これまでのクラスと同様にメンバ変数やメソッドを定義することができる。これに加えて，サブクラスではスーパークラスのメンバをあたかもサブクラスで定義したかのように扱うことができる。ただし、スーパークラスで private と宣言されているメンバについては，サブクラスでもアクセスできない。　スーパークラスで定義されているメソッドを，再度，サブクラスで定義することができる。これをオーバーライドと呼ぶ。同じ名前で引数が異なるメソッドを定義するオーバーロードに似ているが，メソッド名と引数の数と型が完全に一致するメソッドを定義できる。このメソッドが呼び出された場合には，新たに定義されたメソッドが有効で，スーパークラスのメソッドを覆い隠す。すなわち，スーパークラスのメソッドをオーバーライドすることで，サブクラスで定義したメソッドに置き換えることができる。
プログラム 8．1 に，Parent クラスを継承して Child クラスを定義する例を示す。この例では，スーパークラスとなる Parent クラスで getName（）と getType（）の 2 つのメソッドを

定義している（5，6 行目）。Child クラスでは，新たに getBaseType（）メソッドを定義し（13 行目），getName（）メソッドをオーバーライドしている（14 行目）。

```
1:/** Parent クラス(スーパークラス)  */
2:class Parent {
3:   String pName = "Parent";            //親の名前
4:
5:   String getName() { return pName; }      //メンバ変数 name を取得する
6:   int getType() { return 1; }              //種別(type)を取得する
7:}
8:
9:/** Parent クラスを継承した Child クラス  */
10:class Child extends Parent {
11:   String cName = "Child";                  //子の名前
12:
13:   int getBaseType() { return 1; }          //Child クラスで新たに定義
14:   String getName() { return cName; }       //Parent クラスのメソッドを
15:                                             //オーバーライド
16:}
17:
18:/** テスト用クラス */
19:class TestInheritance {
20:   public static void main (String argv[]) {
21:      Parent p = new Parent();               //parent クラスのオブジェクトを生成
22:      Child c = new Child();                 //Child クラスのオブジェクトを生成
23:      String name;
24:      int type, baseType;
25:
26:      name = p.getName();
27:      type = p.getType();
28:      System.out.println("Parent name = " + name + ", type = " + type);
29:
30:      name = c.getName();
31:      type = c.getType();
32:      baseType = c.getBaseType();
33:      System.out.println("Child name = " + name + ", type = " + type +
34:                                              ", baseType = " + baseType);
35:   }
36:}
```

プログラム８．１　継承の例（TestInheritance.java）

Child クラスでは getType（）メソッドを定義していないが，Child クラスのオブジェクトに対して getType（）メソッドを呼び出すと，スーパークラスである Parent クラスの getType（）メソッドが呼び出される。また，同じ Child クラスのオブジェクトに対して getName（）メソッドを呼び出すと、今度はオーバーライドしているので，Child クラスで定義した方の getName（）が呼び出される。したがって，プログラム 8．1 の実行結果は以下の通りである。

Parent name＝Parent、　type＝1
Child name＝Child、 type＝1，baseType＝1

8．3　実行時の型

継承では is－a 関係が重要であるということを思い出そう。つまり，サブクラスはスーパークラスの一種であるという関係である。この関係は変数への参照の代入にも当てはまる。すなわち、スーパークラスの型の変数にサブクラスのオブジェクトへの参照を代入する場合、自動的にスーパークラスにキャストされる。

```
Parent　pl＝new　Child（）；　　//　サブクラスのオブジェクトへの参照を代入
```

しかし、逆に、サブクラスの型の変数にスーパークラスのオブジェクトへの参照を代入することはできない[1]。

[1] Child c ＝ （Child new Parent（）; のような強引なキャストでコンパイルエラーは避けることができるが、実行時に例外(ClassException)を発生する。

```
Child cl＝new Parent（）；　//　コンパイルエラー
```

ここで注意しなければならないのは，サブクラスでオレバーライドしたメソッドの呼び出しである。pl に対してオーバーライドしたメソッド（例えばプログラム 8．1 の getName（）メソッド）を呼び出すとどうなるのだろうか。変数 pl は Parent 型であるので Parent クラスの getName（）メソッドが呼び出されるのか，それとも変数 pl が参照する実際のオブジェクトは Child 型なので Child クラスの getName（）メソッドが呼び出されるのであろうか。
Java では，変数の型ではなく，実行時のオブジェクトの型に従ってメソッドの呼び出しが行われる。したがって，この例では，オブジェクトの実際のクラスである Child クラスの getName（）メソッドが呼び出される。

オブジェクトがあるクラスのオブジェクトかどうか調べるには instanceof 演算子を使用する。プログラム 8．1 の Parent クラスと Child クラスのオブジェクトについて，instanceof 演算子を使用したプログラムの例を示す。

```
1:class TestInstanceof {
2:   public static void main (String argv[]) {
3:      Parent p = new Parent();
4:      Child c = new Child();
5:
6:      //p が Parent クラスのオブジェクトか調べる
7:      if(p instanceof Parent)
8:         System.out.println("p is a Parent object.");
9:      else
10:        System.out.println("p is not a Parent object.");
11:
12:     //p が Child クラスのオブジェクトか調べる
13:     if(p instanceof Child)
14:        System.out.println("p is a Child object.");
```

```
15:      else
16:         System.out.println("p is not a Child object.");
17:
18:       //c が Parent クラスのオブジェクトか調べる
19:      if(c instanceof Parent)
20:         System.out.println("c is a Parent object.");
21:      else
22:         System.out.println("c is not a Parent object.");
23:
24:      //c が Child クラスのオブジェクトか調べる
25:      if(c instanceof Child)
26:         System.out.println("c is a Child object.");
27:       else
28:         System.out.println("c is not a Child object.");
29:   }
30:}
```

プログラム８．２　instanceof 演算子（TestInstanceof.java）

プログラム 8．2 の実行結果を以下に示す。Parent クラスのサブクラスである Child クラスのオブジェクトは、Parent クラスのオブジェクトでもあり，Child クラスのオブジェクトでもあることが示されている。

```
p   is   a Parent objeect.
p   is   not a Child object.
c   is   a Parent object.
c   is   a Child object.
```

演習 8．1
プログラム 8．1 を変更して、Parent 型の変数に Child 型のオブジェクトを代入し、getName（）メソッドを呼び出してみよ。

8．4　スーパークラスのメソッドの呼び出し

オーバーライドされたスーパークラスのメソッドは外部から呼び出すことはできなくなるが，サブクラスの中から呼び出すことはできる。
以下では，プログラム 8．1 の Parent クラスと Child クラスを例に説明する。Parent クラスは getName（）メソッドと getType（）メソッドを持っているが getName（）メソッドだけが Child クラスでオーバーライドされている。

```
class Child extends Parent｛
. . .
   int getBaseType（）｛return l ; ｝                //   Child クラスでのみ定義
      String getName（）｛return cName ; ｝     //   Parent クラスのメソッドを
}                                                                  //   オーバーライド
```

サブクラスでは，自分自身で定義したメソッドを呼び出すのと同じように，スーパークラス

のメソッドを呼び出すことができる。例えば，Child クラスの childMethod（）から，Child クラスの getBaseType（）メソッドと，スーパークラスの getType（）メソッドを呼び出す例を以下に示す。

```
/＊＊　Child クラスのメソッド　＊/
void childMethod（）{
  int type＝getType（）;　　//　スーパークラスのメソッド呼び出し
  int parentType＝getBaseType（）;　//Child クラスのメソッドの呼び出し
}
```

さて，Child クラスで getType（）メソッドをオーバーライドして，タイプ番号として 2 を返すように変更したとする。

```
class Child extends Parent｛　...
  int getType｛　return　2 ; ｝　　//　新たに定義。Parent クラスの
                                    //　メソッドをオーバーライド
  int getBaseType（）｛ return　1 ; ｝　//　Child クラスのみで定義
}
```

ここで，getBaseType（）メソッドは Parent クラスのタイプ番号である 1 を返しているので，Parent クラスの getType（）メソッドと実質的に同じである。また，新たにオーバーライドした getType（）メソッドは，Parent クラスの getType（）メソッドの戻り値＋1 を返せばよいことが分かる。しかし，例えば getBaseType（）メソッドで getType（）メソッドを呼び出すと，Child クラスの getType（）メソッドが呼び出される。

```
    class Child extends Parent｛...
      int getBaseType（）
            int baseType＝getType（）;　//これは Child クラスのメソッド
            return baseType ;
      }
}
```

このように，サブクラスでオーバーライドされたメンバに，そのメンバ名だけでアクセスすると，サブクラスのものが優先される。したがって，スーパークラスのメンバにアクセスしたいときは明示的に行う必要がある。サブクラス内で，スーパークラスは予約語 super で表され，スーパークラスのメンバには以下のようにアクセスする。

```
    super．メソッド　　あるいは　　super．メンバ変数
```

Child クラスの getBaseType（）メソッドと getType（）メソッドを，スーパークラスの getType（）メソッドを呼び出すことで，以下のように書き換えることができる。

```
    class Child extends　Parent｛　...
      int getType｛　return super．getType（）＋1 ; ｝
      int getBaseType（）｛　return super．getType（）; ｝
```

}

演習 8．2
プログラム 8．1 の Child クラスの getBaseType（）メソッドを，スーパークラスの getType（）メソッドを利用するように変更しなさい。また，getType（）メソッドをオーバーライドして，スーパークラスのタイプ番号＋1 を返すようにしなさい。

8．5　継承とコンストラクタ

サブクラスのオブジェクトを生成する場合，特に指定しなくてもスーパークラスのコンストラクタも呼び出される。正確には，継承関係の大元のスーパークラスから順にコンストラクタが呼び出され　最後に該当するサブクラスのコンストラクタが呼び出される。

コンストラクタが複数定義されている場合，引数の数と型に従って呼び出されるコンストラクタが決まることは既に述べた。どのコンストラクタを用いてサブクラスのオブジェクトを生成するにせよ，その中で特に指定しなければ，スーパークラスについては引数なしのコンストラクタが自動的に呼び出される [2]。ただし，スーパークラスでコンストラクタが定義されているが，引数なしのコンストラクタだけが定義されていない場合は，呼び出すべきコンスラクタが存在しないためにコンパイルエラーになる。

コンストラクタから自分のクラスの別のコンストラクタを呼び出すとき，this（）を使用したことを思い出していただきたい。

```
class MyClass｛...
    MyClass（int x）｛
        this（x，0）；　//コンストラクタ MyClass（int x，int y）を呼び出す
  }
  ...
}
```

同様に，スーパークラスの引数ありのコンストラクタを呼び出す場合は予約語 super（）を使用する。スーパークラスのコンストラクタの中で，super（）に渡される引数の型と数に一致するものが呼び出される。ただし，super（）はサブクラスのコンストラクタの先頭でだけ実行できる。

[2]　スーパークラスでコンストラクタを定義していない場合は，自動的に生成されるデフォルトのコンストラクタが呼び出される。

```
class Child｛...
    Child（int x）｛　//　スーパークラスのコンストラクタの中で
        super（x）；　//　int 型の引数をひとつとるコンストラクタを呼び出す
        ...
    }
}
```

8．6　抽象クラス

スーパークラスは，サブクラスに対して，より一般的な概念であると考えることができる。

したがって，継承が何段にもわたるような場合，スーパークラスのそのまたスーパークラスというように，上位のクラスになればなるほど，抽象的な概念になっていく。

ある程度抽象度が上がると，確かにそのような操作や手続きは存在するが，サブクラスでないと具体的に定義できないメソッドが出てくる。例えば，テープレコーダ，CD プレーヤ，ビデオの共通のスーパークラスとして，「（何かを）再生するもの」を表す Player クラスを定義したとする。Player クラスは「再生する」ためのメソッド（play（）メソッド）を持つことは確実だが，Play（）メソッドが具体的に何をするかはそのサブクラス，つまり，テープレコーダなのか，CD プレーヤなのか，ビデオなのかが決まらないと定義できない。

このように，具体的な中身を決めることができないメソッドは，予約語 abstract を付けて，メソッドの名前と型についてだけ宣言することができる。このようなメソッドを抽象メソッドと呼ぶ。以下に，Player クラスに抽象メソッド play（）を宣言する例を挙げる。

```
public class Player｛. . .
    public abstract void play（）;
    . . .
}
```

抽象メソッドには具体的な処理が記述されていないため，そのままでは呼び出すことができない。すなわち，抽象メソッドは，いずれどこかのサブクラスでオーバーライドされて初めて使用できる。ただし，必ずしも直接のサブクラスでオーバーライドしなければならないわけではない。

抽象メソッドを持つクラスのオブジェクトを生成することはできない。このようなクラスは抽象クラス（abstract class）と呼ばれ，やはり，クラス宣言の際に予約語 abstract を使用して明示することができる。

```
abstract class Player｛. . .｝
```

8．7　final クラス

あるクラスを継承の末端としたい，すなわち，あるクラスからこれ以上派生させたくない場合，予約語 final を使用して以下のようにクラスを宣言する。

```
final class foo｛. . .｝
```

final の付いたクラスはスーパークラスになることはできない。すなわち，継承して新しいサブクラスを作成することはできない。

final をメソッドだけに付けることもできる。以下に final メソッドの例を挙げる。

```
public  class  foo｛  . . .
        public final void func（）｛. . .｝
        . . .
}
```

final メソッドを持つクラスは，（そのクラス自体が final クラスでない限り）継承することができるが，サブクラスで final メソッドをオーバーライドすることはできない。

8．8　インタフェース

そのオブジェクトが何であるかは問題ではなく，どのような操作ができるかだけが問題となることがある。すなわち，オブジェクトに対するメソッド，一般的に言うところのインタフェースさえ決めておければ十分という状況である。これを抽象クラスによって実現すると，そのクラスは抽象メソッドだけを持ち，メンバ変数は一切持たないクラスになる。このような場合，Java では抽象クラスを宣言する代わりに，インタフェースを宣言することができる。インタフェースには，抽象メソッドと定数だけを定義することができる。インタフェースの宣言は，予約語 interface を使用して，以下のような形式をとる。

修飾子 interface インタフェース名｛…｝

例えば，インタフェース Movable が，画面に描画できる図形などで，ある座標に移動させるメソッド moveTo（）を定義するものであるとすると，その宣言は以下の通りである。

```
Public　interface Movable｛
        Public void moveTo（int x，int y）；
｝
```

インタフェースの宣言はクラスの宣言とよく似ているが，インタフェース自身は抽象型であり，インスタンス化することはできない。また，宣言のボディには，抽象メソッドの宣言と，定数（final　static）の宣言だけができる。

クラスで継承ができたように，インタフェースも別のインタフェースを継承することができる。インタフェース NewInterface がインタフェース Movable を継承する場合は，以下のようにする。

Public　interface Newlnterface extends Movable｛. . .｝

クラスを継承する際は，クラス宣言で extends を用いてスーパークラスを指定したが，実装するインタフェースを示す場合には予約語 implements を使用し，実装するインタフェースをカンマ“,”で区切って列挙することができる。

class MyShape implements Movable，Erasable｛. . .｝

実装するインタフェースを指定して，そのインタフェースのメソッドを定義しないこともできる。その場合は，宣言したクラスは抽象クラスとなる。

インタフェースの代わりに抽象クラスを用いることもできなくはないが，問題が発生する場合がある。Java では複数のスーパークラスを持つことができず，上記の例のように複数のインタフェースを実装するようなことは，抽象クラスでは実現できない。