(4) ユーザ定義型

抽象データ型を定義できるプログラミング言語で，ユーザが定義した任意の型をユーザ定義型と呼ぶ。以下は，仮想的なプログラミング言語を用いた複素数型の定義例である。なお，演算を定義できない言語でも，データ型だけは定義できる言語（例：Microsoft Visual Basic）もある。これらの言語でも，ユーザが定義した任意の型をユーザ定義型と呼ぶ。

[例]　複素数型の定義例

```
type complex
{ float r;                          /* 実数部の型*/
  float i;                          /* 虚数部の型*/
  complexConst (float a, float b)   /* 複素数の定数設定 */
    { complex  c;  c.r = a;  c.i = b; return©;}
  Operation = complex c             /* 複素数の代入定義 */
    { r = c.r;  i = c.i; }
  Operation + complex c             /* 複素数の加算定義 */
    { r = r + c.r;  i = i + c.i; }
  Operation - complex c             /* 複素数の減算定義 */
    { r = r - c.r;  i = i - c.i; }
  Operation * complex c             /* 複素数の乗算定義 */
    {   float a, b;
        a = r * c.r - i * c.i; /* 実数部 */
        b = r * c.i + i * c.r; /* 虚数部 */
        r = a;  i = b;
     }
  Operation / complex c             /* 複素数の除算定義 */
    {   float a, b, S;
        S = c.r * c.r + c.i * c.i;
        a = (r * c.r + i * c.i) / S ; /* 実数部 */
        b = (i * c.r - r * c.i) / S ; /* 虚数部 */
        r = a;  i = b;
     }
 }
```

以上の定義により，たとえば，次のように記述することができる。

[例]　複素数型の記述例

```
complex c1, c2, c3, c4;
{  c1 = complexConst(1.0, 2.0);       /* 実数部 1，虚数部 2 の定数 */
   c2 = c1 + complexConst(3.0, 4.0);  /* c1 に複素数の定数を加える*/
   c3 = c1 * c2                       /* 乗算                     */
   c4 = c3 / c1                       /* 除算                     *
 }
```


3 オブジェクト指向型プログラミング言語

(1) オブジェクト指向型プログラミング言語とは

オブジェクト指向型プログラミング言語とは，オブジェクト指向の考え方に沿った機構を持つプログラミング言語である。代表的なオブジェクト指向プログラミングとしては，Smalltalk，C++，Java 等がある。

(2) 差分プログラミング

オブジェクト指向では，ものをオブジェクトとしてモデル化し，基本的にはその汎化関係でクラス関係を構成する。プログラミングにおいても，この考え方を踏襲する。すなわち，プログラム構造そのものもオブジェクト間の関係として構成し，これをクラスという関係で表現する。

汎化関係では，上位クラスに定義された変数やメソッドは，上位クラスから下位クラスに継承される。したがって，上位クラスの定義を下位クラスで定義する必要がないので，下位クラスでは上位クラスと異なる部分だけを定義すればよい。これを**差分プログラミング**と呼ぶ。

(3) メッセージパシングとメソッド

通常の手続き型プログラミング言語では，何らかの計算を行うには，関数やサブルーチンにパラメータを与えて，関数やサブルーチンで実行を行う。一方，オブジェクト指向では，オブジェクト（データのかたまり）に対して，何らかのメッセージを与え，オブジェクトの反応としての戻り値を得る。このことをメッセージパシングという。

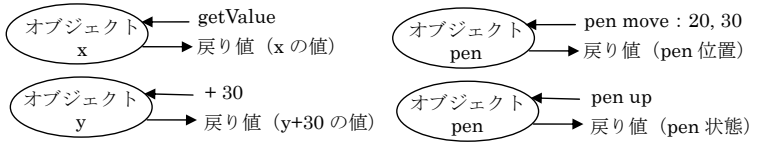


図 7-2　メッセージパシングの考え方

メッセージパシングによるプログラミング上の記述は，例えば以下のようになる（以下は，C++または Java による記述である）。

[例] Java，C++による記述例

```
x.getValue();     y = y + 30;
pen.move(20, 30); pen.up();
```

なお，このメッセージパシングを，通常の手続き型言語で解釈すると，オブジェクト専用の関数（メソッド）を持ち，メッセージ到着により専用の関数が実行される考えてもよい。

（4）メソッドのアクセス可能範囲

通常の手続き型プログラミング言語では，モジュールの中だけで有効な名前（Private, Local 等）と，プログラム全体にわたって有効な名前（Public, Global, Common など）がある。オブジェクト指向プログラミング言語においても同様，メソッドのアクセス可能範囲が指定可能である。アクセス可能範囲でメソッドを分類すると，次のように分けることができる。

- **公開メソッド**（Public Method）：ユーザに公開するメソッド。
- **保護メソッド**（Protected Method）：サブクラスからの継承によってのみアクセスできるメソッド。
- **私的メソッド**（Private Method）：クラス内でのみアクセスできるメソッド。

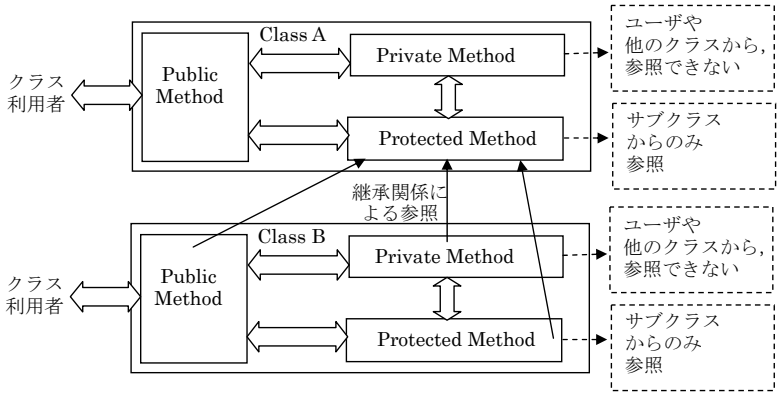


図 7-3　メソッドのアクセス可能範囲

4 論理によるプログラミング言語

（1）論理的な推論規則による実行

推論規則（推論ルール）をあらかじめ定義しておき，ある結論が成立するかどうかを推論する過程は，計算過程そのものであるとする考え方がある。この考え方により，基本的には 1 階述語論理による推論機構を持ったプログラミング言語が Prolog（Programming in Logic）である。なお，実際の Prolog 処理系では，拡張性のため高階述語も用意するのが一般的である。

手続き型言語では，プログラムの処理手順はすべてプログラマが指定するが，Prolog では，推論規則だけを指定し，推論を進めるための実行制御は推論機構が決定する。実行順序を自分で指定しないため，手続き型言語に慣れた者にとっては，実行状態を把握することが困難である。しかし，実行順序を考えないで論理的な推論規則を定義して実行確認することができるので，プログラミング前のアルゴリズム確認に用いられることも多い。

（2）論理的な知識表現

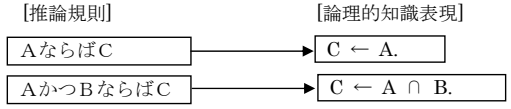
論理的な知識表現は，基本的に次の 2 つの知識で表現する。

① **事実**：ものやできごとの間に成立する関係。たとえば，太郎は，花子と一郎と次郎の父であり，太一の父は和男であり，太一の母は花子であるという事実を，図 7-4 のように表現する。

```
父(太郎，一郎).
父(太郎，花子).
父(太郎，次郎).
父(和男，太一).
母(花子，太一).
```

図 7-4　論理的知識表現における事実の記述例

② **推論規則**：事実からひとつの結論を導く方法を次のように記述する。



例えば，父または母であれば親であり，親の父は祖父であるという推論