オブジェクト指向プログラミングとは?

- 以下の機能や特徴のいくつか、あるいは多くを活用したプログラミング技法
  - カプセル化(振る舞いの隠蔽とデータ隠蔽)
  - インヘリタンス(継承) -- クラスベース
  - ポリモフィズム(多態性、多相性)
  - ダイナミックバインディング(動的束縛) -- 動的型付け

(Wikipediaによる)

Fortran 2003のオブジェクト指向向け機能 例

- カプセル化（振る舞いの隠蔽とデータ隠蔽)
  - モジュール中で，PRIVATE属性の指定が可能。
  - モジュールや派生型中で，データと関連する手続を定義できる。
- インヘリタンス（継承） -- クラスベース
  - ある派生型をベースに，別の派生型を定義できる(EXTENDS)
- ポリモフィズム（多態性、多相性）
  - CLASSにより，実際の型を実行時に決められる
- ダイナミックバインディング（動的束縛） -- 動的型付け
  - 該当なし

Fortran 95のプログラム例

パラメタ

```
module param
integer :: nx,ny,nt
real(8),parameter :: dens=...
real(8) :: gosa
    :
contains
  subroutine setparam(n,err)
   ！各種パラメタの設定
  end subroutine
end module
```

主要データとソルバ

```
module solver
  use param
  real(8),allocatable,dimension(:,:) :: vx,vy,p,...
  private :: vx,vy,p, ... ! モジュール外から参照不可
  contains
   subroutine init(n,err)
     setparam(n,err)
     allocate(vx(nx+1,ny),vy(nx,ny+1),p(nx,ny),...)
          :
   end subroutine
   subroutine pressure()
      求解処理
   end subroutine
end module
```

主プログラム

```
program main
use solver

call init(100,1d-10)  ！初期化
do it=1,nt  ！時間発展ループ

   call pressure()

enddo
```

- モジュールsolverは，データ 及び それを操作する手続をひとまとめにして，「カプセル化」している
- しかし，これをオブジェクト指向プログラミングとは通常言わない
  - 「オブジェクト」がない (vx,vyなどの変数やpressureなどの手続はーつしか存在しない)

オブジェクト指向的に書いた例(1)

ソルバのモジュール

```
module mod_solver
  type :: jacobi ！求解のための派生型
    real(8):: gosa
    contains
    procedure,pass :: solve！求解手続の宣言
  end type
 contains
  subroutine solve(this,....)！求解手続本体
    class(jacobi) ,intent(in) :: this
      求解処理
  end subroutine
end module
```

データとソルバのモジュール

```
module mod_capsule
   use mod_solver
   type, public :: capsule
     real(8),allocatable,dimension(:,:) :: vx,vy,p,...
     class(jacobi),allocatable :: solver
     contains
        procedure,public,pass::construct
        procedure,public,pass::pressure
   end type
   contains
       :
     subroutine construct(this,solver)
       class(capsule),intent(inout) :: this
       class(jacobi),intent(in) :: solver
       ！ 求解手続の派生型の割付け
       !「多相性」の例：引数によって
       !   異なるソルバを割り付ける
       allocate(this%solver, source=solver)
     end subroutine
      ！ソルバを呼び出すサブルーチン
     subroutine pressure(this,nt)
       class(capsule) :: this
       do it=1,nt
         call this%solver%solve(...)
       enddo
     end subroutine
end module
```

オブジェクト指向的に書いた例(2)

主プログラム

```
program main
  use mod_solver
  use mod_capsule, only:capsule
  type(capsule):: fluid ！データとソルバの派生型
   :
  ！ソルバを派生型構成子で設定
  call fluid%construct(solver=jacobi(gosa=1d-10))
  ！ 実際の計算
  call fluid%pressure(nt)

end
```

- これは，「オブジェクト」
  内部のデータやソルバは，オブジェクト毎に別の実体となる

オブジェクト指向的に書いた例(3)

- 内部処理(求解)が変わっても，同じインタフェイスで書ける
  - 別のソルバは，別のオブジェクトとして共存可能
  - 勿論，ソルバを置き換えることも容易

別に作成したソルバのモジュール

```
module mod_sor
  use  mod_solver
  !「継承」の例
  type,extends(jacobi) :: sor !  jacobiを拡張してsorを宣言
    contains
    procedure,pass :: solve ! 別の求解手続で上書きする
  end type
  contains
    subroutine solve(this,....)
    class(sor), intent(in) :: this
     求解処理
    end subroutine
end module
```

主プログラム

```
program main
  use mod_solver
  use mod_capsule, only:capsule
  use mod_sor, only:sor
  type(capsule):: fluid ！データとソルバのオブジェクト
  type(capsule):: fluid2 ！データとソルバの別のオブジェクト
   :
  ！ソルバを派生型構成子で決める
  call fluid%construct(solver=jacobi(gosa=1d-10))
  ! 実際の計算
  call fluid%pressure(nt)

  call fluid2%construct(solver=sor(gosa=1d-10))
  call fluid2%pressure(nt)
end
```

利点と欠点

- メリット
  - 保守性・生産性の向上
    - 実装の中味を変えても，インタフェイスを同じに保つことができる(使い方を変えなく済む)
      - 例えば，ソルバを改造してもプログラム本体は僅かな修正で済む
      - 実行時に，ソルバを切り替えることも容易
    - 各モジュールの独立性が高いので，モジュール毎に，別の人が作ることも容易
      - 大規模なコードや，ライブラリの開発では重要
- デメリット
  - 記述量は，(少なくとも最初の開発時は)多くなりそう
    - 改造時の記述量は少なくできる(はず)
  - 性能は，落ちる可能性が高い (良くなることはない)
    - 自動的にメモリの割付け・解放が発生することによるオーバヘッド・最適化の抑制
    - 手続呼出しによるオーバヘッド
      - 処理が細切れになり，最適化の範囲が狭まる
        - 特に呼び出す手続が翻訳時に決まらない場合，インライン展開等，手続をまたがる最適化ができなくなる
      - privateな定数・変数を関数経由で取得するオーバヘッドや最適化の抑制
    - 派生型の利用により，メモリアロケーションが変化し，性能に影響する

コンパイラによる自動的なメモリ割付けの例

```
subroutine sub(x,y,z)
real(8) :: x(n),y(n),z(n)
real(8), allocatable a(:), b(:), c(:)
   :

a=x
b=y
c=z
```

コンパイラによる変換イメージ

```
if(aとxの形状が異なる)then
  再割付け(a(xと同じ形状))
endif
do i=lbound(x,1),ubound(x,1)
  a(i) = x(i)
enddo
if(bとyの形状が異なる)then
  再割付け(b(yと同じ形状))
endif
do i=lbound(y,1),ubound(y,1)
  b(i) = y(i)
enddo
if(cとzの形状が異なる)then
  再割付け(c(zと同じ形状))
endif
do i=lbound(z,1),ubound(z,1)
  c(i) = z(i)
enddo
```

- Fortran 95での変換イメージ

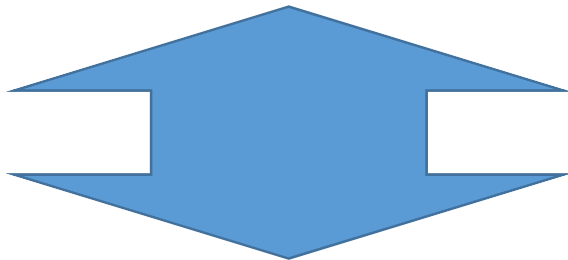
```
subroutine sub(x,y,z)
real(8) :: x(n,n),y(n,n),z(n,n)
real(8), allocatable a(:), b(:), c(:)
   :

do i=lbound(x,1),ubound(x,1)
  a(i) = x(i)
  b(i) = y(i)
  c(i) = z(i)
enddo
```

- Fortran 2003では，右辺と左辺の形状が異なる場合，左辺は自動的に再割付けされる
  - チェックや再割付け等の関連する処理により，(例え，実際には再割付けがおきなかったとしても)最適化が抑制される
- 以下のように書けば大丈夫(回避する方法はある)

```
a(:)=x(:)
b(:)=y(:)
c(:)=z(:)
```

Fortranはどこへ向かうべきか?

- 現代的な機能を積極的に取り入れるべき?
  - 性能は「多少は」犠牲にしても生産性を向上させるべき
  - 数値計算だけでなく，より守備範囲を広げることでFortranは復権する
    - コンパイラに実装されるまで時間がかかっても待てる



- Fortranは，性能を最優先に追求すべき?
  - 数値計算の専用言語で良い
    - 性能に特化したサブセットが欲しい (処理系依存の要素があるので現実には難しい)

生産性・保守性と性能のバランスは?

生産性の向上と引き換えに，許容できる性能低下はどの程度?

- 10~20%の性能低下なら?
- 50%の性能低下なら?
- 80%の性能低下なら?