Java によるオブジェクト指向プログラミング
今おさえておきたいオブジェクト指向の三大要素

千葉県立千葉商業高等学校教諭　**鵜野澤　博**

1．はじめに

新学習指導要領「プログラミング」では，改訂のポイントとして「オブジェクト指向型言語や手続き型言語など，指導するプログラム言語に応じて指導内容を選択できる幅を拡大」とある。「オブジェクト指向型言語」とは，Java を指すものであり，各都道府県における研修会等では，Java の講習会を実施していることと思われる。

これまで手続き型言語である COBOL やイベント駆動型のプログラミングを指導してきた我々にとって，「オブジェクト指向」というものを正しく理解し，指導できなければならない状況になってきた。現在，ソフトウェア開発の現場では，「オブジェクト指向」がトレンドであり，Java に限らず，C++，C#，VisualBasic.NET などオブジェクト指向を採用したプログラミング言語が主流である。なぜ現場では，「オブジェクト指向」が主流なのだろうか。

ソフトウェア開発においては，ニーズの変化に伴い，それまでの考え方や手法では限界を迎えるようになってきた。つまり，多様なビジネスシーンで利用するプログラムの開発においては，手続き型・構造化プログラミングでは柔軟に対応できない状況になってきた。仕様の追加や変更に柔軟に対応でき，さらに開発期間を短縮することが求められるようになってきたため，それを可能にするオブジェクト指向が広く採用されるようになったのである。しかし，従来の構造化プログラミングが，オブジェクト指向プログラミングにすべて変わったのではない。構造化プログラミングのメリットも周知のとおりであり，オブジェクト指向プログラミングがベストだと判断した時にオブジェクト指向を採用すべきである。そのためには，オブジェクト指向を正しく理解していなくてはならないのである。

2．オブジェクト指向プログラミングとは

これまで，あるデータをコンピュータで処理する場合に，コンピュータが行う「作業」に着目し，コンピュータがやるべき作業をその手順に沿ってそのままプログラミングを行ってきた。このように，まず「データ」があり，それに対してコンピュータにやらせる「作業」（＝処理）をプログラミングするということから手続き型と呼んでいる。

また，プログラミングした作業の中で，共通化できる部分をまとめ，再利用性を高めて開発を効率化していくのが構造化プログラミングである。共通化した部品をサブルーチンや関数と呼ぶ。

一方，オブジェクト指向では，やりたいことの中で登場する「現実にあるモノ」に着目し，それぞれのモノが持つ「データ」とそのモノが備える「振る舞い」（＝処理）をまとめ，できる限りそのままプログラム化していくスタンスを取る。ここで，同じ「処理」という意味の「作業」と「振る舞い」をどのように区別するのだろうか。「作業」とは，「データ」に対して定義・作成されるものであり，コンピュータの視点に立ったものである。一方「振る舞い」とは，「現実にあるモノ」（＝オブジェクト）に対して定義・作成されるものであり，人間の視点に立ったものと言える。

オブジェクト指向プログラミングでは，このようにコンピュータでやりたいことをオブジェクトとして分析し，「データ」と「振る舞い」をいくつかのオブジェクトごとにまとめた後，各オブジェクト同士がお互いにメッセージをやり取りしながら，やりたいことを実現していく。この「振る舞い」は，そのオブジェクト自身が備えている可能な動作のことであり，他のオブジェクトから「あなたの備えている動作を実行してください」と依頼された時に，実行することになる。その依頼をメッセージのやり取りと表現した。

3．オブジェクト指向プログラミングのメリット

具体的にオブジェクト指向プログラミングのメリットは何であろうか。そのメリットとしては，

(1) やりたいことをプログラム化しやすい
(2) プログラムの再利用性が高い
(3) 機能の追加・変更がしやすい
ことが考えられる。そのため，ソフトウェア開発や保守を効率よく柔軟に行えるのである。

(1) やりたいことをプログラム化しやすい

求められるプログラムは，人間が利用するものであり，本来人間がやるべきことを代行するものである。つまり，プログラムで扱うモノは人間にかかわるモノであり，現実にあるモノということになる。オブジェクト指向プログラミングは，やりたいことを現実にあるモノを観点にし，人間の視点でプログラムを考え，まとめていくため，やりたいことをプログラム化しやすいといえる。

(2) プログラムの再利用性が高い

オブジェクト指向プログラミングでは，「データ」と「振る舞い」をまとめたオブジェクトというプログラムのパーツを組み合わせて，プログラムを形作る。この「データ」と「振る舞い」をひとまとめにしたオブジェクトを Java においては，「クラス」と呼ぶ。この「クラス」という単位でプログラムを組むことからプログラム化しやすく，かつ再利用性が高いといわれる。また，再利用性の高さには次のような種類が考えられる。

ア．同じプログラムのパーツ（＝クラス）を幅広く使い回せること。
イ．過去に作ったクラスから，同じクラスを簡単に作れること。
ウ．過去に作ったクラスから，似たクラスを簡単に作れること。

このような再利用性の高さを実現するものが，クラスを基礎とした，オブジェクト指向の三大要素といわれる「カプセル化」，「継承」，「ポリモーフィズム」である。また，この三大要素により機能の追加・変更にも強いのである。

(3) 機能の追加・変更がしやすい

ソフトウェア開発の現場では，機能の追加・変更は日常茶飯事である。この機能の追加・変更がしやすいこととは，具体的には次の３つである。

エ．プログラムのパーツ１つ１つの依存性が低いこと。
オ．他のプログラムのパーツと幅広く組み合わせられること。
カ．ソースを書き換える部分が少なく済むこと。

依存性が低いということは，プログラムのパーツを追加・変更した際，その影響が他のパーツに及ぼされないことである。他に影響を及ぼしてしまうとプログラム全体の動作に思わぬ不具合が生じる危険性が高くなる。依存性が低ければ，その影響をパーツ内だけに封じ込めておける。これを実現できるのが「カプセル化」である。しかし，依存性が低いだけではどうしようもない。プログラムを形作っている他のパーツとうまく結合できなければならない。他の多くの種類のパーツとうまく結合できるのであれば，組み合わせ可能な範囲が広くなり，機能の追加や変更がより柔軟に行えるのである。また，依存性が低ければ，機能の追加・変更に伴うソースコードの書き換えも基本的にはそのパーツ内だけになる。これらを実現できるのが「継承」であり，「ポリモーフィズム」である。

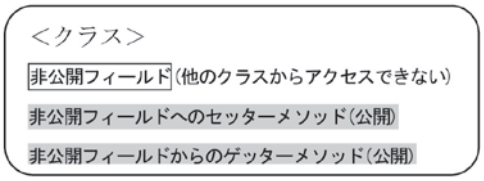
４．カプセル化

カプセル化（encapsulation）とは，「カプセルに詰めること」を意味する。オブジェクト指向プログラミングでは，あるオブジェクトが持つ複数の「データ」（＝フィールド）と複数の「振る舞い」（＝メソッド）を１つにまとめ，クラスとして記述する仕組みを指す。単にまとめるだけでなく，フィールドやメソッドにアクセスできる対象を特定の範囲内に限定することも可能なのである。つまり，カプセル化は，フィールドとメソッドをまとめ，クラスというソフトウェアのパーツを作り，かつアクセス範囲を限定して各パーツの独立性を高める仕組みなのである。なぜこのようなことをするのだろうか。

オブジェクト指向では，クラス内のフィールドへの直接アクセスを許さず，何らかのメソッドを通してアクセスさせることがよいと考えられている。オブジェクトの中身を他のオブジェクトからは見えないようにし，間接的にアクセスする手段を用意してアクセスさせるようにするのである。これを「アクセサメソッド」という。

このアクセサメソッドの中で，フィールドに入っている値を取得するメソッドを「ゲッターメソッド」，フィールドに値を設定するメソッドを「セッターメソッド」と呼ぶ。活用方法としては，ゲッターメソッドのみを用意して，フィールドを読み取り専用にする。これは，他のクラスでフィールドの値を書き換えられたくない場合によく用いられる。ま

た，セッターメソッドの中にチェック機能を持たせて，代入される引数の値の妥当性をチェックしてから代入するようにできる。このように必要に応じて，セッターメソッドとゲッターメソッドを用意して活用することでプログラムが安全に処理でき，さらにクラスの独立性も高まるのである。



もちろん，クラスを作成しただけでは，プログラムは何も処理をしない。つまり，１つのクラスは，あるオブジェクトが持つフィールドと備えるメソッドをまとめたものであり，言い換えれば，単なる「設計図」である。その設計図だけでは，そのクラスを利用することはできない。設計図を基に実体となるオブジェクトを生成しなければ利用できない。クラスから生成された実体（オブジェクト）のことを「インスタンス」といい，インスタンスを生成することを「インスタンス化」という。１つのクラスから複数のインスタンスを生成することができる。例外として，インスタンスを生成しなくても利用できるクラスがある。それは，プログラムの開始点となるクラスである。そのクラスは，main（ ）メソッドを持つクラスであり，プログラムのすべての出発点となる。

５．継承

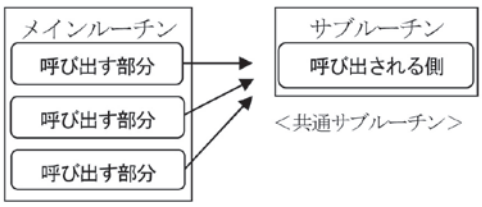
継承（inheritance）とは，あるクラスのメソッドやフィールドなどの定義情報を自分のクラスに引き継ぐ仕組みである。例えば，過去に作成したプログラムのパーツに対して，追加や変更を加え，新しいプログラムのパーツを作成できるようにする。基になるクラスを「親クラス」「スーパークラス」「基底クラス」と呼び，引き継ぐクラスを「子クラス」「サブクラス」「派生クラス」と呼ぶ。

この継承をうまく利用すると同じコードが複数の場所に存在しないようにプログラミングできる。例えば，よく似た機能を持つクラスを２つ作るとき，クラスを１つ作り，そのコードをコピーしてクラスの名前を書き換えれば実現できる。しかし，この場合は同じコードが複数の場所に存在することになり，もしコピー元に修正箇所が出たときには，コピー先のコードも修正しなくてはならない。しかもコピー先で独自の変更を行っている場合もある。そうなると，どこをどのように修正すべきかクラスごとに考えなければならない。こうしたことを防ぐ意味でも継承を利用するメリットがある。また，似たクラスを複数作ることが最初からわかっているならば，同じ部分を抽出し，その部分を親クラスとして定義して継承させることができる。こうすることで，より効率的なプログラミングができる。

しかし，このように便利な継承も注意して利用する必要がある。不適切な継承や，多階層に継承することは避けなければならない。不適切な継承とは，スーパークラスの中にサブクラスでは必要のないフィールドやメソッドを含んで継承をしてしまうことである。また，多階層に継承することで，あるクラスの変更が継承関係にある他のクラスへ影響し，バグの原因が分かりにくくなる恐れがある。これでは，追加変更に強いとは逆に言えなくなってしまう。そのため，クラスを作るときには，スーパークラスからサブクラスを安易に作るのではなく，本当に必要なクラスがいくつあるのかを考え，それらのクラスから共通点を探し，その共通点だけを実装したスーパークラスを作り，そのクラスを継承するようにすべきである。

６．ポリモーフィズム

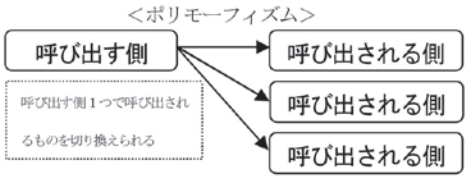
ポリモーフィズム（polymorphism）は，日本語では「多態性」と訳すのが一般的である。この仕組みは，「呼び出す側を共通化する」ものである。これまでの手続き型言語では，呼び出される側を共通化する共通サブルーチンのみであった。これは，呼び出す側がいくつ増えても，呼び出される側を修正する必要がないものである。



それに対し，ポリモーフィズムは，呼び出す側を共通化し，呼び出される側が増えても，呼び出す側を修正する必要がないものである。

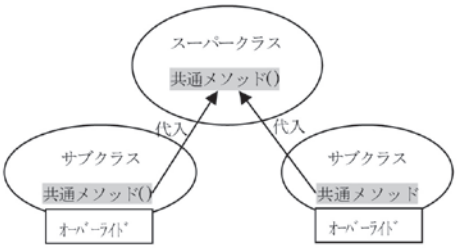
具体的には，同じ名前のメソッドが，そのオブジェクトのクラスに応じてそれぞれ適切な処理を行え

るようにする仕組みである。現在，フレームワークと呼ばれる大規模な再利用部品があるのもこの仕組みの賜物といえる。



(1) 継承を利用したポリモーフィズム

継承とは，いくつかのクラスに共通する部分を括りだし，スーパークラスとして独立させ，それを継承し，拡張してサブクラスとして利用することだった。この場合，サブクラスはスーパークラスの１つでもある。つまり，サブクラスのインスタンスは，スーパークラスの変数でも扱えることを意味し，サブクラスの変数をスーパークラスの変数に代入できるのである。スーパークラスの変数を媒介にして，代入するオーバーライドしたサブクラスの変数を切り換えることでポリモーフィズムが実現できる。

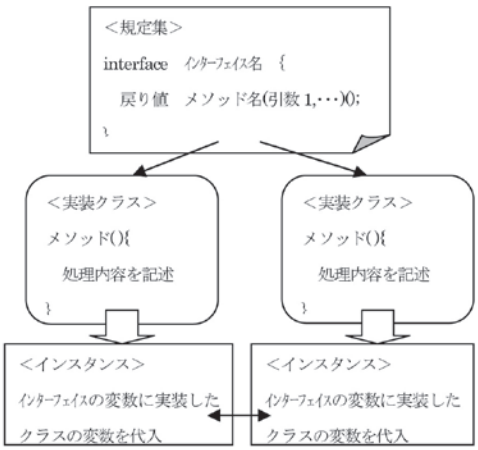


(2) インターフェイスを利用したポリモーフィズム

インターフェイスとは，「そのオブジェクトはこのような振る舞いを持っている」という「規定集」のようなもの。つまり，そのオブジェクトが持っている外部とつながるための窓口（境界面）を定めたものである。

規定されているのは，あくまでもメソッドの名前，引数，戻り値だけであり，メソッドの処理内容は規定されていない。つまり，処理内容については，そのメソッドを定義しているオブジェクト（クラス）におまかせなのである。

インターフェイスは，クラスのようにインスタンスを生成できない。必ず別のクラスに「実装」して使うことになる。インターフェイスを実装したクラスは，必ずその中で，そのインターフェイス内で規定したメソッドの中身を記述しなければならない。また，インターフェイスの変数を宣言して，実装したクラスの変数を代入できるため，このインターフェイスの変数を媒介として複数のクラスのメソッドを切り替えることでポリモーフィズムを実現する。



継承を使う場合とインターフェイスを使う場合では何が違うのであろうか。インターフェイスを使うメリットは，「継承を使わずに済む」ことである。継承は，スーパークラスを元にして，サブクラスを簡単に作成できる。それゆえに，クラス同士の結びつきが強くなり依存性が高くなる。依存性の高さは，オブジェクト指向のメリットである「追加・変更への強さ」が得られないことを意味する。スーパークラスの下に何層もサブクラスを作ってしまうと，スーパークラスのちょっとした変更により正しく動作しないサブクラスができてしまうのである。

それに対して，インスタンスを生成できないインターフェイスを使うと，不用意な継承が行われなくなり，依存性が高まることは少なくなる。

Java では，継承は１回につき１つのクラスしかできないようになっている。しかし，インターフェイスは，複数同時に実装できる。このことは，実装クラスのインスタンスを他のインスタンスと幅広く組み合わせて使えるようになるため，再利用性の向上や追加・変更への強さを達成できるのである。

７．おわりに

科目「プログラミング」において，オブジェクト指向言語である Java を採用する場合は，オブジェクト指向プログラミングを意識して指導する必要がある。今回は紹介できなかったが，実際にコードを入力して，オブジェクト指向をコードから理解する必要がある。今後も研究していきたい。