9　オブジェクト指向に関連する用語群

本章では、Java 言語が踏襲している「オブジェクト指向」という考え方と、それに関連する典型的な用語をいくつか紹介します。

9.1　オブジェクト指向

オブジェクト指向 とは、プログラムが処理するデータ（数値）や処理手順を、現実世界の「モノ」になぞらえた考え方のひとつです。

例えば 1.1 以降、私たちは何度か、BMI を算出するプログラムを使用してきました。いま、このプログラムを「BMI 電卓」という「モノ」にたとえてみましょう。皆さんはこの電卓を実現するときに、どんな部品が必要になりますか？おそらく多くの人は、以下の 3 種類の部品が必要だと考えるでしょう。

- 体重や身長を入力するためのボタン部品。
- BMI を計算するための計算部品。
- BMI を計算した結果を出力するための表示部品。

このように、「モノ」はいくつかの部品で構成されます。いままで習得してきた Java 言語では、ちょうど「モノ」がクラスに相当し、「部品」がメソッドや変数に相当する、と考えることができます。

では、複数の「モノ」を組み合わせて使うことを考えてみましょう。例えば折れ線グラフを描く装置と BMI 電卓を組み合わせると、長期間にわたる BMI 値の変動を描くことができるでしょう。それと同様に、Java 言語では、多くのクラスを組み合わせることで、さらに高度な処理を実現できます。

このように、プログラムによる処理を、「モノの組み合わせ」として考えることが、「オブジェクト指向」の基本的な考え方となります。

皆さんは Java 言語以外の言語で、**フローチャート**などの形で処理の手順を描く機会があるかもしれません。オブジェクト指向は、処理手順に基づいて組み立てる旧来のプログラム言語に対比する新しい考え方として成長した、という歴史的経緯のある考え方です。

カプセル化

上述のとおり、本章ではオブジェクト指向を BMI 電卓に比喩して説明しています。ここで BMI 電卓という「モノ」を、「目に見えるモノ」と「目に見えないモノ」に分類して考えてみましょう。あくまでも一例として、以下のように分類できるのではないでしょうか。

目に見えるモノ： 身長や体重の入力部品、BMI 算出結果の出力部品。

目に見えないモノ： BMI 計算式、BMI 計算途中の値。

こうしてみると、モノの中と外での情報のやり取りは目に見えるように、そしてモノの内部処理は目に見えないように設計するのが一般的であろう、と考えることができます。

Java 言語でもこれと同様に、クラスの中と外で情報をやり取りする部分は積極的に他のクラスやパッケージに公開し、クラスの中で閉じた形で用いる部分は他のクラスやパッケージに公開しない、という考え方が普及しています。このような考え方を **カプセル化** と呼びます。

7.4 節にて public と private というキーワードを説明しましたが、これらのキーワードはまさに、カプセル化を実現するために、積極的に公開するメソッドや変数と、公開しないメソッドや変数を指定するためのキーワード、といえるかと思います。

9.2 派生と継承：extends

Java 言語では、クラスに「親子」という概念を持ち込むことができます。そして、ある「親クラス」の性質（具体的にはメソッドや変数）を「子クラス」にも持たせることができます。このことを「親クラスから子クラスへの**継承**」といいます。また、このときの子クラスを「親クラスから**派生**したクラス」といいます。

派生と継承を実現するために、Java 言語では extends というキーワードを用意しています。

一例として、以下のように Parent という親クラスを定義したとします。

```
public class Parent {
   public void calcSquare(double a) {
     double square = a * a;
     System.out.println("Square=" + square);
   }
}
```

続いて一例として、以下のように Child という子クラスを定義します。

```
public class Child extends Parent {
   public void calcCube(double a) {
     double cube = a * a * a;
     System.out.println("Cube=" + cube);
   }
}
```

ここで public class Child の次にある「extends Parent」に注目してください。これは、Child クラスは「Parent クラスの派生クラス」であり、「Parent クラスの性質を継承するクラス」であり「Parent クラスを拡張するクラス」であることを示しています。具体的には、Child クラスにも Parent クラスのメソッドである calcSquare が継承されて使えるようになります。

この Child クラスを呼び出す main メソッドの例を、以下に示します。

```
public class UseChild {
   public static void main(String[] args) {
     double a = 3.0;
     Child child = new Child();
     child.calcSquare(a);
     child.calcCube(a);
   }
}
```

この main メソッドでは、Child クラスの変数 child のメソッド calcSquare および calcCube を呼び出しています。calcSquare は元々Parent クラスのメソッドですが、Child クラスがこれを継承しているので、あたかも Child クラスのメソッドであるかのように使うことができます。

例えばBMI電卓の新商品が、旧商品の機能を受け継いだまま、新しい機能を追加搭載したとしましょう。このとき、旧商品を親、新商品を子と考えると、旧商品の機能は親から子に受け継がれた上で、新商品には新機能が搭載される、と考えられます。Java言語における派生と継承の考え方は、このような形で現実世界の「モノ」にも共通する考え方である、といえます。

9.3 オーバーライド：final と abstract

Java言語では逆に、親クラスから派生した子クラスにて、あえて親クラスのメソッドと同名のメソッドを新しく定義しなおすことができます。このようにして新しく定義することを、**オーバーライド** といいます。

一例として、以下のように Parent という親クラスを定義したとします。

```
public class Parent {
   public void print() {
     System.out.println("I am the Parent.");
   }
}
```

続いて一例として、以下のように Child という子クラスを定義します。

```
public class Child extends Parent {
   public void print() {
     System.out.println("I am the Child");
   }
}
```

Childクラスは Parentクラスの派生クラスなので、本来でしたら Parentクラスの printメソッドを継承するはずです。しかし、この Childクラスでは、printメソッドをオーバーライドしているため、Parentクラスの printメソッドは実行されずに、Childクラスの printメソッドが実行されます。

このオーバーライドという機能は、一例として以下のような状況にて有効に活用できます。仮に Parentクラスに10個以上のメソッドがあったとします。これの派生クラスである Childクラスでも、Parentクラスの大半のメソッドを継承したままで使いたいのですが、どうしても1,2個だけ、メソッドの中身を書き換えたいとします。このとき、Parentクラスとの継承関係を絶つよりも、その1,2個のメソッドだけを書き換えたほうが、プログラム開発において格段に合理的です。このような状況において、オーバーライドは有効に活用できるといえます。

final

親クラスのメソッドに **final** と定義することで、子クラスによるオーバーライドを禁止することができます。

一例として、以下のように Parent という親クラスを定義したとします。

```
public class Parent {
   public final void print() {
     System.out.println("I am the Parent.");
```

```
    }
}
```

このとき print メソッドには final が指定されているので、Parent クラスを継承する派生クラスにて print メソッドを再定義してしまうと、コンパイル（javac 命令の実行）にてエラーを生じてしまいます。

abstract

親クラスのメソッドに **abstract** と定義することで、子クラスによるオーバーライドを強制することができます。

一例として、以下のように Parent という親クラスを定義したとします。

```
public abstract class Parent {
   public abstract void print() {
   }
}
```

このとき print メソッドには abstract が指定されているので、Parent クラスを継承する派生クラスにて print メソッドが再定義されていないと、コンパイルにてエラーを生じてしまいます。

なお、abstract が指定されたメソッドを含むクラスは、クラス自体にも abstract 指定をしないといけません。上述の例では、abstract 指定された print というメソッドを含む Parent クラスは、クラス自体にも abstract 指定しないといけないので、public abstract class Parent というように記述されています。

このように、abstract 指定されたクラスやメソッドを、**抽象クラス** あるいは**抽象メソッド** と呼びます。

Java 言語では、抽象クラスや抽象メソッドなどを用いて、とりあえず抽象的にクラス名やメソッド名を定義しておき、メソッドの具体的な中身は継承クラスにまかせる、というような技法がよく用いられます。これは例えば、プログラムを 2 人以上の人で開発するときに、最初の担当者がとりあえず抽象クラスを定義しておき、次の担当者が継承クラスを開発する、というような分担作業をするときに便利です。

9.4 仕様と実装：interface と implements

Java 言語では、前節で紹介した抽象クラスとは別に、**インタフェース** という定義に基づいて、抽象的にクラスやメソッドを定義することができます。

ここで早速ですが、インタフェースの例を示します。

```
public interface Parent {
   public void print();
   public void calcSquare(double a);
}
```

いままでクラスを定義するときには、1 行目はお決まりのように public class というような単語が並んでいましたが、インタフェースを定義するときには class という単語の代わりに **interface** を使います。

そしてインタフェースでは、メソッドの名前、引数、返り値は定義するだけで、その中身（中カッコで括られた処理本体）は一切記述しません。

では、その中身はどのように定義すればいいのでしょうか。以下にその例を示します。

```
public class MyParent implements Parent {
   public void print() {
     System.out.println("I calculate Square.");
   }
   public void calcSquare(double a) {
     double square = a * a;
     System.out.println("Square=" + square);
   }
}
```

このプログラムの 1 行目に着目してください。1 行目の前半 `public class MyParent` は、今まで示した多くのプログラムと同様に、このクラスが `MyParent` というクラスであることを示しています。続いて 1 行目の後半 `implements Parent` は、`Parent` というインタフェースの中身を定義していることを示します。このように、インタフェースで定義されたメソッドの中身を記述するクラスには、**implements** という単語を用いて、「どのインタフェースの中身を記述したか」を指定します。

このようなインタフェースとクラスの対比的な関係を、しばしば**仕様** と**実装** という単語で表現することがあります。「仕様」とは多くの場合、規格や規則を定めたものを指します。Java 言語の場合、メソッドの名前、引数、返り値などの規格が仕様に相当します。そして、その処理本体が実装に相当します。

Java 言語に限らず、多くのプログラム言語の開発では、まず仕様が策定され、続いてプログラマは実装を開発します。これは BMI 電卓にたとえると、まずデザインや機能が決められ、続いてその機能を実現するための中身（電子回路など）を開発する、という手順に似ています。

Java 言語が採用している `interface` および `implements` の機能は、現実世界にも浸透している「仕様と実装」という考え方を実現しやすくした、といえます。

また、インタフェースは抽象クラスと同様に、複数の人でプログラムを開発するときに、非常に有用です。例えば最初の開発者が、とりあえずインタフェースを定義しておき、次の担当者がその中身を記述するクラスを開発する、というような分担作業をするときに便利です。インタフェースと抽象クラスは以上の意味で役割が類似していますが、強いて違いをあげるとしたら、インタフェースは仕様だけを定義する仕組みであるのに対して、仕様だけでなく実装の一部を継承したり再定義したりする、というようなことが必要な場合に抽象クラスを活用します。

9.5 【サンプルプログラム】複数の相性占い師

それでは、「仕様と実装」の考え方を端的に表したプログラムの例を示します。6.4 節で紹介した「相性占い」のプログラムを大きく拡張したものです。かなりプログラムが大きくなりますが、頑張ってください。

まず、下のインタフェースを、`Uranaishi.java` というファイルで保存してください。このインタフェースは、2 人の月日を代入して、相性を返すメソッドを定義します。

皆さんがプログラムの発注者になって、いろんな人に相性計算のプログラムを書いてほしい、という状況を想像してください。このとき、

下のインタフェースをプログラマに公開するだけで、プログラマたちは相性計算のメソッドの名前、引数、返り値を知ることができます。 これによって、いろんなプログラマが、統一された形式で相性計算の

プログラムを書くことができます。

```
public interface Uranaishi {

   // 2 人の月日を代入して、相性を返すメソッドを定義する
   public int calculateChemistry(int month1, int day1, int month2, int day2);
}
```

続いて、上のインタフェースにしたがって、2 人のプログラマが相性計算のプログラムを開発した、という状況を想像してください。

まず Alice さんが、インタフェース Uranaishi の記述にしたがって、UranaishiAlice というクラスを開発したとします。このプログラムは「implements Uranaishi」と書いてあることからもわかるように、インタフェース Uranaishi の記述にしたがって相性計算のプログラムを書いています。

では、下のプログラムを、UranaishiAlice.java というファイルに保存してください。

```
public class UranaishiAlice implements Uranaishi {

   // 元旦からの合計日数を求める
   int calculateTotalDays(int month1, int day1) {
     int total = 0;
     int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

     // 誕生月の前の月までの合計日数を求める
     for(int i = 0; i < (month1 - 1); i++) {
       total += days[i];
     }

     // 合計日数に日を足す
     total += day1;

     // 合計日数を返す
     return total;
   }

   // 2 人の月日を代入して、0〜100 パーセントで相性を求める
   public int calculateChemistry(int month1, int day1, int month2, int day2) {
     int chemistry = 0;

     // 人物 A の誕生日の元旦からの合計日数を求める
     int total1 = calculateTotalDays(month1, day1);
     // 人物 B の誕生日の元旦からの合計日数を求める
     int total2 = calculateTotalDays(month2, day2);

     // 相性の算出式 （当てずっぽうな算出式なので信頼できるものではない）
     chemistry = (total1 + total2) % 101;
```

```
    // 相性を返す
    return chemistry;
  }
}
```

続いて同様に Bob さんが、インタフェース Uranaishi の記述にしたがって、UranaishiBob というクラスを開発したとします。

では、下のプログラムを、UranaishiBob.java というファイルに保存してください。

```
public class UranaishiBob implements Uranaishi {

  // 2 人の月日を代入して、0〜100 パーセントで相性を求める
  public int calculateChemistry(int month1, int day1, int month2, int day2) {
    int chemistry = 0;

    // 人物 A の月と日を足す
    int total1 = month1 + day1;
    // 人物 B の月と日を足す
    int total2 = month2 + day2;

    // 相性の算出式 （当てずっぽうな算出式なので信頼できるものではない）
    chemistry = (total1 + total2) * 100000 % 101;

    // 相性を返す
    return chemistry;
  }
}
```

そして最後に、かつて作成した Uranai.java を UranaiMulti.java にコピーして、以下のように書き換えてください。

```
import java.io.*;

public class UranaiMulti {

  // メインメソッド
  public static void main(String[] args) {

    // 2 人の誕生日の月と日を代入する変数
    int month1 = 0, month2 = 0, day1 = 0, day2 = 0;

    // 相性の算出結果を代入する変数
    int chemistry = 0;
```

```
      // 人物 A の月、日、人物 B の月、日、の順に標準入力する
      try {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        System.out.println("Month of birthday A");
        month1 = Integer.parseInt(br.readLine());
        System.out.println("Day of birthday A");
        day1 = Integer.parseInt(br.readLine());
        System.out.println("Month of birthday B");
        month2 = Integer.parseInt(br.readLine());
        System.out.println("Day of birthday B");
        day2 = Integer.parseInt(br.readLine());
        br.close();
      }
      catch (Exception e) {
        System.out.println(e);
      }

      // Alice による相性算出結果を表示する
      UranaishiAlice u1 = new UranaishiAlice();
      chemistry = u1.calculateChemistry(month1, day1, month2, day2);
      System.out.println("By Alice:  chemistry= " + chemistry);

      // Bob による相性算出結果を表示する
      UranaishiBob u2 = new UranaishiBob();
      chemistry = u2.calculateChemistry(month1, day1, month2, day2);
      System.out.println("By Bob:  chemistry= " + chemistry);

   }
}
```

これを実行すると、例えばこんな感じで、Alice と Bob による 2 人の相性計算結果が出力されます。

```
Month of birthday A
1
Day of birthday A
3
Month of birthday B
4
Day of birthday B
12
By Alice:  chemistry= 4
By Bob:  chemistry= 99
```

このプログラミングのミソは、Alice と Bob が 2 人とも、インタフェース Uranaishi にしたがって相性計算のプログラムを書いていることです。これによって、UranaiMulti クラスでは、Alice と Bob の相性計算を、そっくりな形で呼び出すことができるので、プログラミングが簡単になります。

ここまでできた人は、近くの友人たちで集まって、以下の課題も挑戦してみてください。

1. 自分だけの相性計算のクラスを作ってみてください。例えば Takayuki という名前の人が作成する相性計算のクラスを、UranaishiTakayuki という名前のクラスにしてみてください。Alice とも Bob とも、また周囲の友人とも違う、自分だけの計算式を考えてプログラミングするのが重要です。
2. 友人どうしで、メールで送り合うか、あるいは USB メモリで交換するなどして、相性計算のクラスを共有してください。
3. UranaiMulti クラスに、Alice や Bob だけでなく、いろんな人による相性計算のクラスをたくさん追加して、いろんな相性計算結果を出してください。

Java 言語の面白い点の一つとして、インタフェースさえ公開すれば、いろんな人が互いに利用し合えるプログラムを開発しやすくなる、という点があります。例えば相性計算であれば、いろんな人による相性計算のプログラムを集めて、信用できる相性計算だけを採用する、時と場合によって相性計算を入れ替える、などといった多彩な楽しみ方が考えられるかと思います。