第回 オブジェクト指向 (演習)

ゼミ担当者 ： 王 路易，吉形 允晴，松井 勇樹，渡辺 章人
指導院生 ： 山川 望，上田 祐一郎
開催日 ： 2007 年 5 月 22 日

1 クラス

クラスとは，オブジェクト作成の基盤となるテンプレートである．基本的にクラスには，オブジェクトの持つ性質であるデータと機能を表すメソッド (命令) で構成され，1 つのオブジェクトの特徴を表現する．Java では，以下のような構造により表現される．

クラスの構造

```
class クラス名
{
   変数型:フィールド変数

   コンストラクタ
   {                              }

   戻り値型:メソッド名 (引数リスト)
   {                              }
}
```

例えば，車というクラスで，性質として「馬力」や「車種」，機能として「走る」を持っているとする．この場合下記のように表現できる．

クラスの構造例

```
class car{
   int power       //性質：馬力
   String model       //性質：車種

   car(){            //コンストラクタ
      power=100;
      model="crown";
   }
   void run(){           //機能：走る
      System.out.println("走る")
   }
}
```

1.1 クラスのインスタンス化

クラスは単にオブジェクト作成のテンプレートに過ぎない．そのためクラスの持つ性質や機能を実際に操作する場合，クラスを 1 つのオブジェクトとして宣言する必要がある．これをクラスのインスタンス化という．Java によって宣言する方法には以下の 2 つの方法がある．

インスタンス化 1

```
クラス名 変数名;
変数名 = new クラス名 ();
```

インスタンス化 2

```
クラス名 変数名 = new クラス名 ();
```

このように,「new」を用いてクラスから 1 つのオブジェクトを作成する．このオブジェクトを変数に代入することにより，下記のような記述でオブジェクトのメソッドやデータの参照を可能とする．

オブジェクトのメソッド利用

```
変数名. メソッド名 ();
```

オブジェクトの変数利用

```
変数名. メンバ変数名;
```

実際に car クラスをインスタンス化すると次のように表現される．同時に実行例も下記に示す．

インスタンス化：変数とメソッドの利用例

```
class main{
   public static void main(String args){
       car car1=new car();
       System.out.println(car1.model);
       car1.run();
   }
}
```

インスタンス化の実行例

```
crown
走る
```

1.2 コンストラクタ

コンストラクタとは，クラスをインスタンス化する際の初期化を意味する．Java では，クラス名と同様のメソッド名にすることにより，それを可能とする．

コンストラクタの記述例

```
class car{
                :
   car(){           //コンストラクタ
      power=100;
      model="crown";
   }
                :
}
```

例えば「car car1=new car()」が実行されたと同時に，このコンストラクタも実行され，「power=100」「model=”crown”」という car オブジェクトが作成されることになる．

1.3 オーバーロード

クラスは引数の異なる同じ名前のメソッドを複数持つことができる．インスタンス化されたメソッドの実行は「変数名. メソッド名 ()」で実行できるが，() 内に引数を入れることにより，その違いを識別することができる．これにより同名のメソッドでも引数により異なった処理を選択できる．その記述例が以下の通りである．

オーバーロードの記述例

```
class car{
//コンストラクタ
  car(){
      power=100;
      model="crown";
  }
  car(int bariki,String syasyu){
      power=bariki;
      model=syasyu;
  }
//メソッド
  void run(){           //機能：走る
    System.out.println("走る")
  }

  void run(int jisoku){
   System.out.println(jisoku+"km/s 走行")
  }
}
```

オーバーロードはコンストラクタにも適応可能である．これは 1 つの例であるが，car オブジェクト宣言時に引数として数値と文字列を入れることにより，その値を用いて power と model を決定できる．また，run メソッドに関しては，int 型の数値を引数とすることで時速を指定して走行速度を決めることができる．といったような使い分けが可能である．

2 修飾子

修飾子とは，クラス・メソッド・コンストラクタにアクセス権限を与えるものである．この修飾子により，クラスの外にメソッドを公開するかどうか，継承を許すかどうかといったコントロールが可能となる．代表的な修飾子を以下の Table1 に示す．

Table 1　修飾子一覧

修飾子	クラス	インタフェース	メソッド	コンスラクタ	変数
public	○	○	○	○	○
protected	○	○	○	○	○
private	○	○	○	○	○
static	○	○	○	×	○
final	○	×	○	○	○
abstract	○	○	○	×	×

これらの修飾子は大きくアクセス修飾子，スタティック修飾子，ファイナル修飾子，抽象修飾子に分けられる．これについては，次に述べる．

2.1 アクセス修飾子 (public，protected，private)

指定した変数やクラスなどを，どの範囲から参照可能かのスコープを制御するのに用いられる．アクセス修飾子のアクセス制限を Table2 に示す．

Table 2　アクセス修飾子

アクセス修飾子	アクセス制限
private	同一クラス内
省略	同一パッケージ内
protected	同一パッケージ内，またはそのサブクラス
public	制限なし

2.2 スタティック修飾子 (static)

メンバ変数，メソッドを宣言する際，static 修飾子が付与されたものをクラスメンバ変数，クラスメソッド，static 修飾子が付与されていないものをインスタンスメンバ変数，インスタンスメソッドと言い，両者で性質が異なる．

インスタンスメンバ変数，インスタンスメソッド

インスタンスメンバ変数，インスタンスメソッドはオブジェクトごとに存在します．クラスからオブジェクトが生成されるとそれぞれのオブジェクトに対し，インスタンスメンバ変数，インスタンスメソッドが割り当てられる．その概念図を Fig. 1 に示す．

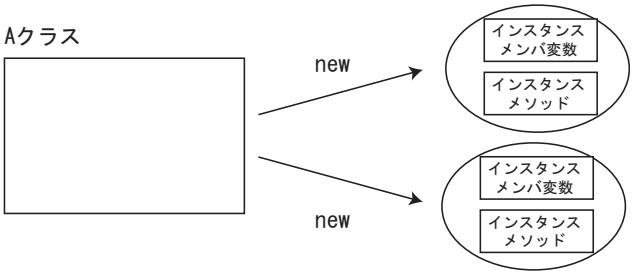

Fig. 1　インスタンスメンバ変数、インスタンスメソッドの位置付け

クラスメンバ変数，クラスメソッド

クラスメンバ変数、クラスメソッドはそのクラス内で１つという位置付けを持った，変数，クラスである．その概念図を Fig. 2 に示す．

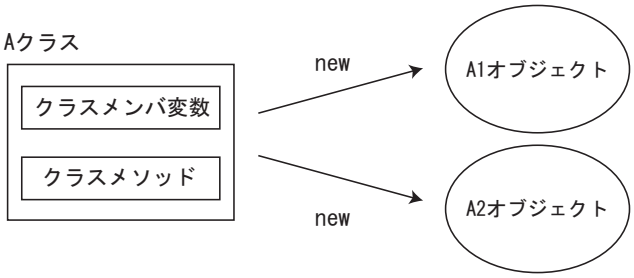

Fig. 2　クラスメンバ変数、クラスメソッドの位置付け

クラスメンバ変数，クラスメソッドはオブジェクトに依存しないため，オブジェクト指向という観点から言えば利用する機会は少ない．

利用されるのは以下のケースである．

- オブジェクト間で共通の値を保持したい場合
- 定数を宣言する場合
- 公式的な処理を行うメソッド
  例)　Math クラスの sin()，cos() メソッド
- オブジェクトを生成せずに処理を行う必要のあるクラスメソッド　例)　main メソッド

2.3　ファイナル修飾子 (final)

クラスに用いた場合はサブクラスを定義できず，メソッドに用いた場合はサブクラスでメソッドをオーバーロードできない．また，変数に用いた場合，その変数は変更できない．そのイメージを Fig. 3 に示す．

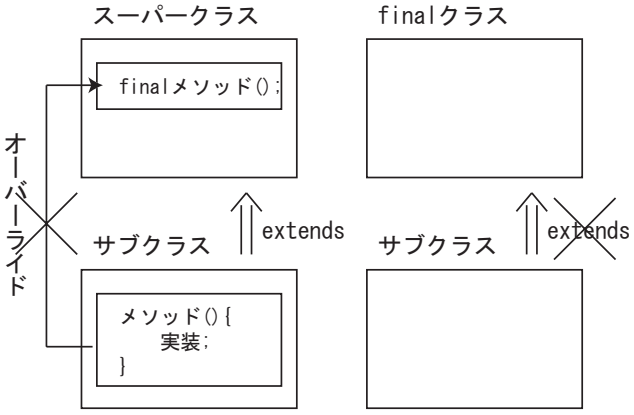

Fig. 3　fainal 修飾子のイメージ

final を使用する理由の一つとしてセキュリティ上の問題がある．ハッカーたちの一つの手段としてあるクラスのサブクラスを作成し，そのサブクラスに不正な動作を行う処理を記述すると言うものがある．これを未然に防ぐためにサブクラスの作成を不可能にする．また，もう一つに理由としては，クラス設計時に，サブクラスを作る必要がないと明示的に宣言することが挙げられる．

2.4　抽象修飾子 (abstract)

抽象クラス（abstract クラス）･抽象メソッド（abstract メソッド）の宣言を行う時に付与される．サブクラスが必ず実装しなくてはならない機能を明確にしておくのが主な利用目的となる．よって抽象クラスは，共通の機能を表現し，個々が持つ独自の機能はそれぞれのサブクラスで実装したい場合に使用する．抽象クラスのイメージを Fig. 4 に示す．

抽象クラスの基本書式

```
abstract　class　クラス名 {　　内容
}
```

抽象メソッドの基本書式

```
abstract　戻り型　メソッド名 ();
```

3　継承

継承とは，あるクラスの性質や機能を別のクラスに引き継ぐ (使用可能とする) ことである．一般的に継承されるクラスをスーパークラスといい，継承するクラスをサブクラスという．Java で継承を表す基本書式は次の通りである．

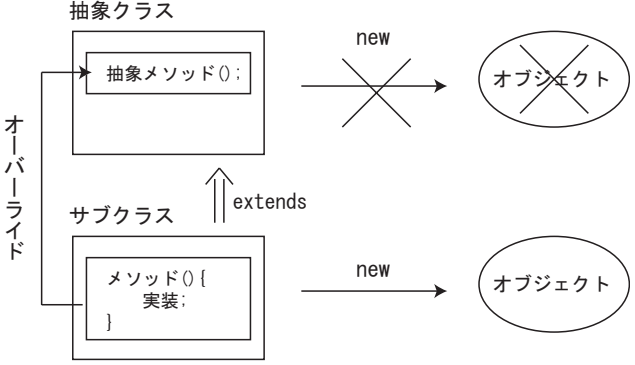

Fig. 4　抽象クラスの概念図

継承の基本書式

```
class サブクラス名 extends スーパークラス名{
        内容
}
```

このように，クラス名を定義すると同時に「extends スーパークラス名」と記述することで継承関係が成り立つ．それにより，サブクラスでスーパークラスのデータやメソッドを，インスタンスなどの宣言をせずとも使用可能となる．

継承のサンプルソース

```
//メイン
class superclass_Demo {
    public static void main(String[] args) {
           plane plane1=new plane(); //サブクラスの実体化
           plane1.fly();              //サブクラスのメソッド
          plane1.move();           //スーパークラスのメソッド
    }
}

//スーパークラス
class norimono {
    void move() {
        System.out.println("移動する");
    }
}

 //サブクラス
class plane extends norimono {
    void fly() {
        System.out.println("飛ぶ");
    }
}
```

継承の実行結果

```
飛ぶ
移動する
```

メインプログラムでは，まず plane を実体化し,plane の fly メソッドを実行している．次に plane のメソッドとして,move メソッドが実行されているが,plane に move メソッドが特別記述されているわけではない．しかし，実行結果として「移動する」が得られている．これは「extends」により，plane が norimono クラスを継承していることを意味している．　これにより，プログラム上で同じ特徴を持つものに同様の機能を持たせる状態を表現できる．

3.1　オーバーライド

オーバーライドとは，継承時にスーパークラスで定義されたメソッドと同じ名前，引数を持つメソッドを再定義することである．スーパークラスのメソッドを変更することはできないが，サブクラスに特化した機能を付与する必要がある場合などに利用する．

オーバーライドの基本形

```
class スーパークラス{
修飾子1 戻り型　メソッド名　(引数型　引数名)
{
     処理1;
}
}
class サブクラス extends スーパークラス{
修飾子2 戻り型　メソッド名　(引数型　引数名)
{
     処理1;
処理2;
}
}
```

条件

- オーバーライドする側はオーバーライドされる側と戻り型、インスタンスメソッド名、引数型、引数の数が同じでなければならない．
- オーバーライドされる側のインスタンスメソッドに指定されるアクセスレベルより厳しい制限を持つアクセスレベルをオーバーライドする側のインスタンスメソッドに付与することはできない．
- オーバーライドされる側のメソッドに final 修飾子が付与されている場合，そのメソッドをオーバーライドすることはできない．

オーバーライドのサンプルソース

```
class override_Demo {
public static void main(String[] args) {
      taxi taxi1=new taxi();  //サブクラスのインスタンス化
      taxi1.run();            //run メソッドの実行
   }
}
//スーパークラス
class car {
    public void run(){
        System.out.println("走る");
    }
}
//サブクラス
class taxi extends car{
    public void run(){
        System.out.println("走るけどお金は取るよ");
    }
}
```

オーバーライドの実行結果

```
走るけどお金は取るよ
```

メインプログラムでは，まず taxi クラスを実体化している．次に taxi クラスの run メソッドを実行している．car と taxi は継承関係し共に run メソッドを保持しているが，taxi は car を継承する際に run メソッドを上書きすることになり，実行結果として taxi の run メソッドが優先されていることがわかる．

4 集約

クラスが他のクラスの組み合わせで構成されている関係を集約という．例えば，「車がエンジンを含む」といったように，車というオブジェクトがエンジンというオブジェクトで構成されている関係 (has-a 関係) が考えられる．集約する側のクラスを Java で記述すると以下のようになる．

Java による集約構造

```
class クラス A{
      クラス B の実体：インスタンス B

    メソッド{
          インスタンス B. メソッド

    }
}
```

上記のクラスは，まず集約する側であるクラス A が，集約される側であるクラス B を内部でインスタンス化することにより，「クラス A がクラス B を含む」という関係が作られている．次に，クラス A のメソッドで，集約されるクラス B のメソッドが実行できるプログラムを定義している．

JAVA では，このようにクラス A がクラス B の実体を含み，かつ，クラス A でクラス B のメソッドが操作できる状態を構築することにより，集約という関係を表現する．

集約プログラム例

```
//メインクラス
public class syuuyaku_Demo {
public static void main(String[] args) {
        car car1=new car(120);
        car1.run();
}

//car クラス
class car{
  //engine クラスのインスタンス化
       engine engine1=new engine();
       int real_speed;

  //コンストラクタ
       car(int max){
         real_speed=0;
       }
  //run メソッド
       void run(){
      //engine クラスの kadou メソッド
         real_speed=engine1.kadou();
         System.out.println("エンジン稼動");
         System.out.println
           ("時速"+real_speed+"km/s");
       }
}

//engine クラス
class engine {
       int kadou(){
          return 10;
       }
}
```

集約の実行結果

```
エンジン稼動時速 10km/s
```

このプログラムは，car クラスが engine クラスを集約していることを表している．car クラスには engine クラスの実体が存在し，car クラスのメソッド内部で engine クラスのメソッドを実行している．

5 インタフェース

インタフェース は，ある機能を実現するクラスが必ず実装するべきメソッドの名前や型といったヘッダー部分のみを宣言したものである．インターフェースを実装したクラスは，そのインタフェースで定義されるメソッド（抽象メソッド）を実装しなければならない．

インタフェースの基本書式

```
修飾子 interface インタフェース名 (extends
  スーパーインタフェース名){
    :
}
```

インタフェースの実装方法

```
修飾子 class クラス名 implements インタフェース名{
インタフェースで記載された内容を実装した処理
}
```

インタフェースのサンプルプログラム

```
interface HelloWorld {

  String MESSAGE = "Hello World";

  void sayHello(int x);
}

public class InterfaceDemo implements HelloWorld {
  public void sayHello(int x) {
    for (int i = 0; i < x; i++) {
      System.out.println("Hello!!");
    }
  }
  public static void main(String[] args) {
    System.out.println(MESSAGE);
    InterfaceDemo obj = new InterfaceDemo();
    obj.sayHello(5);
  }
}
```

インターフェースの実行結果

```
Hello World
Hello!!
Hello!!
Hello!!
Hello!!
Hello!!
```

インターフェースの実行結果-未実装

```
Exception in thread "main" java.lang.Error:
コンパイル問題が未解決です。
型 InterfaceDemo は継承された抽象メソッド
HelloWorld.sayHello(int) を実装する必要があります。
```

6 パッケージ

パッケージとは，クラス (インターフェース含む) を区別しやすくするために，階層構造を持たせる仕組みである．Java では，クラスファイルを保存するディレクトリをパッケージと定め，以下の効果を実現している．

- 同じ役割を持つクラス・インタフェースを１つのパッケージにまとめることにより，そのクラス
- インタフェースの持つ意味がわかりやすくなる．
- 同じ名前を持つクラス・インタフェースが複数ある場合，名前の衝突を避けることができる．
- クラス・メンバ変数・メソッド・コンストラクタにアクセス制限をつけることができる．

パッケージ宣言

```
package パッケージ名
```

パッケージの利用

```
import パッケージ名. クラス名
```

サンプルプログラム

```
-----------------------------------------
package packageA;

public class Pack {  //(2)
  public void showBelong() {
    System.out.println("PackageA");
  }
}


-----------------------------------------
package PackageB;

public class Pack {
  public void showBelong() {
    System.out.println("PackageB");
  }
}


-----------------------------------------
import PackageA.Pack;

public class Package {
  public static void main(String[] args) {
    Pack ex = new Pack();
    ex.showBelong();
  }
}
```

7 例外処理

例外とはプログラム動作における予期せぬ事象 (エラー) のことを言う．Java ではデフォルトの例外処理として，例外発生時にあらかじめ用意されているエラーメッセージ（Exception in thread ”main” java.lang.ArithmeticException ・・・など) を表示しプログラムを終了する．このデフォルトの例外処理以外にエラーメールの送信，エラーログの生成，強制終了処理の実行といった任意の例外処理を付け加えることが可能である．

7.1 例外の種類

Java では例外をクラスにより管理している．すべての例外クラスは Throwable クラスの子孫クラスである．

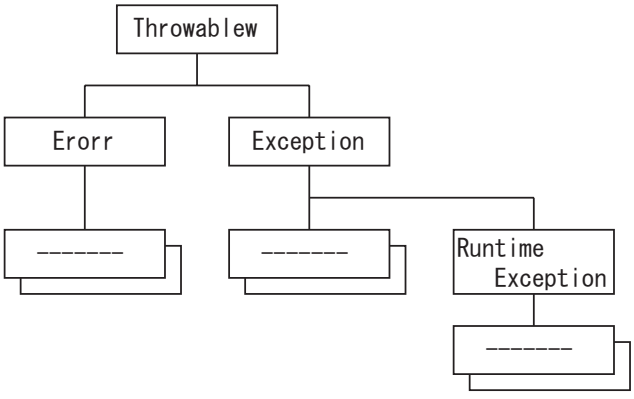

Fig. 5　例外クラス

例外クラスは，Error クラス、RuntimeException クラス、RuntimeException クラス以外の Exception クラスに分類される．Error クラスを継承する例外クラスは，

例外処理では復旧できない例外である．以下に例を示す．

- OutOfMemory Error ……メモリ枯渇

RuntimeException クラスを継承する例外クラスは，JavaVM の内部で発生する例外である．以下に例を示す．

- ArithmeticException……ゼロ除算
- ArrayIndexOutOfBoundsException……不正インデックスによる配列参照

この二つの任意の例外処理を記述することはあまり推奨されておらず，また任意の例外を記述しなくてもデフォルトの例外処理が行われる．そのため，これらの例外は**非チェック例外**と呼ばれる．

一方，RuntimeException 以外の Exception クラスにおいては必ず任意の例外処理を記述しなければならない．記述しない場合，コンパイルエラーが発生する．例外処理が必ず必要なことより，これらの例外は**チェック例外**と呼ばれる．以下に例を示す．

- ClassNotFoundException……未定義クラスの呼び出し
- IOException……入出力処理の失敗

7.2 try，catch，finally

try、catch、finally は Java での例外処理における基本事項である．例外処理を行うための基本書式は以下の通りである．

エラー処理の基本書式

```
try{
    例外をスローする可能性のある処理
}catch(例外クラス型　引数名){
    例外処理 (例外ハンドラ)
}catch(例外クラス型２　引数名){
    例外処理 2(例外ハンドラ 2)
}finally{
    最後に必ず実行される処理
}
```

例外発生時，上記のソースは以下のように振舞う．

1. 例外オブジェクトというエラー情報を保有したオブジェクトをスローする (生成する)
2. スローされた例外オブジェクトが，catch 節にに記載された例外クラスの型と合致した場合，そのキャッチ節に記載された例外ハンドラを実行する．
3. finally 節に記載された処理を実行する．

エラー処理サンプルプログラム

```
public class TryCatchDemo {
public static void main (String[] args) {
try {
     int x = 10/0;
    }catch (Exception e) {
     System.out.println("Error Type：" + e);
    }finally{
     System.out.println("Fainally Execution");
    }
}
}
```

エラー処理の実行結果

```
Error Type：java.lang.ArithmeticException: / by zero
Fainally Execution
```