ソースの構造
クラス
クラスの継承

第1.0版

作　成　者	NH 中村広二
作　成　日	2019年10月23日
最終更新日	2019年10月23日

ソースの構造

Java プログラムのソースファイル全体の構造、コンパイルと実行のプロセス、およびよく使われる Java 用語について説明する。

パッケージ

Java プログラムの構成単位はクラス。クラス名だけで管理していると、すぐに名前の衝突が発生する。 パッケージはこれを防ぐために用意された仕組み。 すなわち関連したクラスをまとめて小包（パッケージ）として管理する。 外部に公開するクラスのみを public として宣言し、内部だけで使うクラスは外から隠蔽する。 外部に公開したクラスを呼び出すには、次のようにパッケージ名の後にピリオド（ . ）を付けて、その後にクラス名を記述する。

```
パッケージ名.クラス名
```

パッケージの階層

パッケージ名自身も名前が衝突しないように、階層化が可能になっている。 例えば

```
java.util.Calendar
```

上記は java パッケージの中の util パッケージの中の Calendar クラスを示している。 これをクラスの完全限定名と言う。 これは次のようにして使う。

```
java.util.Calendar cal = java.util.Calendar.getInstance();      // ☆
```

独自パッケージ

上の例は Java 言語が標準的に提供しているパッケージでしたが、開発者が独自にパッケージを作ることもできる。 それには、Java ソースの先頭に、次のように書く。

```
package mypackage;
```

package は独自のパッケージを宣言する旨のキーワード。 次の mypackage は独自に付けたパッケージ名。 パッケージ名も命名規約にしたがう必要がある。 これによってこのソースファイルに含まれるすべてのクラスが mypackage というパッケージに収納されることになる。

パッケージとアクセス制御

前述のように外部に公開するクラスには public 属性を付ける。 public が付けられるクラスは、1 つの Java ソースファイルで 1 つに限られる。 public 属性はクラスのメソッドにも付けることができる。 これについてはクラスの章で説明する。

パッケージ名の一意性

パッケージ名を階層的に作れたとしても、全世界の人が作る Java のパッケージ名の一意性の保証は困難。 そこで Java の言語仕様は、作成したプログラムを広い範囲に公開する場合はインターネットのドメイン名を使って一意性のあるパッケージ名を付けるように主張している。 abc.co.jp というドメイン名を持つ組織に所属していれば以下のようなパッケージ名を付けることになる。

jp.co.abc.組織内の一意名

インポート文

上の例（☆）で示したように、いつも完全限定名を使っていては表記が長すぎる。 そこでインポートという仕組みが用意された。 まずソースファイルに 01: のように書いておく。 ☆は 02: のように、java.util の部分を省略できる。

```
01:   import java.util.Calendar;
02:   Calendar cal = Calendar.getInstance();
```

また、03: のように * を使うと、java.util 内のすべてのクラスを使うことができる。
ただし、これは java.util の直下にしか有効ではない。 例えば java.util.jar パッケージ内のクラスも使おうとすると、04: のような記述も必要。

```
03:   import java.util.*;
04:   import java.util.jar.*;
```

ソースファイルの構成

ソースファイルの例を示す。 以下は A.java と言うファイル名の内容。

```
--A.java----------------------------------
package mypackage;       // package 文
import java.util.*;      // import 文
import java.io.*;        // import 文
public class A {         // クラス定義
      …
}
class B {                // クラス定義
      …
}
------------------------------------------
```

Java ソースの規約

Java のソースファイルは以下の条件を守らねばならない。

package 文はファイルの先頭に記述する。 package 文は 0 または 1 行である。
引き続いて import 文を記述する。 import 文は 0～複数行記述できる。
次にクラス定義を記述する。
1 ソースファイルに複数のクラスを記述できるが業務で使うときは 1 ソースファイル 1 クラスが望ましい。
public が指定できるクラスは 1 つのソースファイルに 0 または 1 クラスである。
public を指定したクラス名とソースファイル名は同じでなければならない。
ソースファイルの拡張子は .java とする。
ソースファイルに main メソッドが存在しなくてもかまわない。

コンパイルと実行

Java のコンパイルは javac コマンドを使い次のように行う。

```
javac A.java
```

コンパイルが正常に終了すると、Java 独自の中間言語（バイトコード）で書かれた、クラスファイルができる。クラスファイルはソースファイルのクラスの数だけできる。 A.java には 2 つのクラスがあったので次のようなファイルができる。

```
A.class
B.class
```

実行

コンパイルの結果できたクラスファイルは次のようにして起動する。 main メソッドをもたないクラスは起動できない。 このとき、java A.class と指定するのではなく、拡張子 .class は不要ですので注意する。

```
java A
```

パッケージの例

Java が標準的に提供しているパッケージの内、代表的なものを示す。 java.lang は最も基本的なパッケージであり、暗黙的に import されているので、明に import しなくても使うことができる。

java.lang	Java プログラム言語の基本的なクラスを提供
java.io	システム入出力の機能を提供
java.util	基本的なユーティリティを提供
java.sql	データベースにアクセスして処理するための機能を提供
java.net	ネットワークアプリケーションのためのクラスを提供

コメント

Java ではプログラムの中に機能や目的など、後で見るときに役立つメモを書いておくことができる。 これがコメント文。 コメント文の部分では、プログラムは何もしないで読みとばすので、プログラムの動作に何も影響を与えない。

Java のコメント文には次の 3 種類がある。 01: はこれまでも使ってきた。 02: は行内の一部や、逆に長い行に渡ったコメントを記述するのに有効。 03: は 02: の一部ですが、この形式でコメントを書いておくと、javadoc（ジャバドック）というツールを使ってドキュメントを作成することができる。

```
01:   //              行末までのコメント
02:   /* … */         範囲指定型(複数行可;ネストは不可)
03:   /** … */        ドキュメント自動作成用のコメント
```

アノテーション

J2SE5.0 からアノテーションという機能が追加された。 アノテーションもコメントと同じようにプログラムの動作に直接影響を与えない。 ただしコメントと違ってプログラムから読み取ったりコンパイラーに影響を与えたりすることができる。 アノテーションは先頭に @ を付けて表す。 以下にアノテーションの例を示す。

メソッド名の前に以下のようなアノテーションを付加すると、そのメソッドがオーバーライドメソッドであることを示す。 このアノテーションが付いているのにもかかわらず、そのメソッドがオーバーライドしていないとコンパイルエラーになる。 これによりタイプミスなどによるメソッド名や引数の型の間違いを検出することができる。

```
@Override
```

また、クラス名の前に以下のようなアノテーションを付けると、そのクラスが EJB のステートレスセッション Bean であることを示す。 EJB コンテナはこれを認識して対応する処理を実行する。

```
@Stateless
```

アノテーションには引数を取るものもある。 例えばクラス名の前に次のように記述すると、コンパイラーが出す unchecked という警告を抑止（SuppressWarning）することができる。

```
@SuppressWarnings("unchecked")
```

これらはアノテーションのごく一部の例。 アノテーションには多くの種類があり、自分で定義することもできる。

アサーション（表明）

アサーションは実行時にある条件をチェックし、その条件が成立しないと AssertionError 例外を発生させる構文。 この仕組みを使うことでデバッグが容易になる。 アサーションの例を示す。

```
assert x >= 0: "x は正のはず！ x = " + x;
```

この文は変数 x が 0 以上であることを表明している。 実際に 0 以上であれば何も起きないが、そうでないと例外を発生し「:」以降の内容を表示する。 x が -1 の時にこの文が実行されると次のように表示する。
AssertTest.java はこのコードが記述されているファイル名、12 はこの文のファイル内の行位置。

```
Exception in thread "main" java.lang.AssertionError: x は正のはず！ x = -1
at AssertTest.main(AssertTest.java:12)
```

このように、要所要所に assert 文を埋め込んでおき、実行時に実際にそれを確認することで、スムースなデバッグが可能になる。 なお、アサーションを有効にするには実行時に次のように -ea オプションを指定する必要がある。

```
java -ea AssertTest
```

ガベージコレクション（未使用領域の返却）

Java では使われなくなった領域（誰からもポイントされなくなった領域）を探して、共用のメモリ領域（ヒープメモリ）に自動的に返却する処理を行っている。 この仕組みをガベージコレクション（ゴミ集め）と言う。 このため C++では大きな問題となっていたメモリリーク（メモリの開放し忘れ）が Java では発生しにくくなった。

ただし、ある領域が未使用となったからといって、必ずしもガベージコレクションが実行されるとは限らない。ガベージコレクションは、処理の合間をぬって実行される。 ある特定の未使用領域に対してガベージコレクションが実行されるかどうか、またいつ行われるかは、ガベージコレクションを行うプログラムに任されている。

※ メモリリーク問題は Java では大きく改善されたが、まだ次の問題がある。
（1） ガベージコレクションが不定期に実行されるので、突然性能が悪化することがある。
（2） プログラマーが使い終わったメモリの解除そのものを忘れると、やはり返却されない領域が少しづつたまりメモリ枯渇を起こすことがある。

Java 用語集

クラスパス

クラスパスは、Java が使用するクラスライブラリーなどの場所を検索する際の、場所を示すリスト。 クラスパスで指定した範囲で使用するクラスが検索できないと、コンパイルまたは実行時にエラーが表示される。 クラスパスを変更するには、コマンドの -classpath オプションを使用するか、または OS の CLASSPATH 環境変数を使用する。

jar ファイル

jar（ジャーと読む）は Java ARchive の略で、Java で標準的に使用されている圧縮ファイル。 クラスファイルなどが ZIP 形式圧縮されて格納されている。 jar ファイルになっていれば、Java が自動的に展開して実行してくれる。 jar ファイルは「jar」というコマンドで作成できる。

javadoc

javadoc とは Java のソースファイルから Java のリファレンスマニュアルを生成してくれるコマンド。 ソースファイルに決まった形式でコメントを書いておくと簡単にリファレンスマニュアルができる。 ソースを修正したが、ドキュメントは古いままということがおこりにくくなる。

JVM

JVM は Java Virtual Machine（Java 仮想マシン）の略。 JVM はコンパイルの結果できたクラスファイルを、各マシンに固有の形式に変換しながら実行する。 JVM は各プラットホームごとに提供される。 Java の基本方針である Write Once Run Anywhere（一度書くとどこでも動く）を実現するため、このような仕組みになっている。

JDK と JRE

JDK（Java Developer's Kit）には Java プログラムの開発に必要なセット一式が含まれている。 一方、JRE（Java Runtime Environment）は Java の実行に必要なプログラムのセットです。 JDK は JRE を含んでいる。

JavaSE／JavaEE／JavaME

JavaSE は Java Platform, Standard Edition の略で、Java の標準機能セット。末尾に版数を付けて JavaSE8 または単に SE8 のように表す。SE5 までは J2SE5 と呼ばれていた。Java にはこの他にサーバ用のソフトウェア群を含んだ JavaEE(- Enterprise Edition)と、携帯電話などの小型デバイス向けの JavaME (- Micro Edition) とがある。

クラス

クラスは Java の学習で最も重要な概念。 Java のプログラムはクラスで構成されている。 簡単なプログラムは 1 つのクラスで構成され、複雑なプログラムは、互いに関連した複数のクラスで構成されている。 クラスはユーザが独自に作れるデータ型。 クラスの基本的なことがらについて説明する。

クラスの例

まずはクラスの例を示す。 クラスは変数とメソッドとから構成されている。 変数はフィールドと呼ばれることもある。 変数とメソッドをまとめてメンバーと言う。 以下の Person クラスは name および age という変数と、それを処理するメソッド群からなっている。

```
class Person {
  String name = "";             // 変数：氏名
  int age = 0;                  // 変数：年齢

  void setName(String n) {      // メソッド：変数 name に氏名をセットする
    name = n;
  }

  void setAge(int a) {          // メソッド：変数 age に年齢をセットする
    age = a;
  }

  String getName() {            // メソッド：変数 name から氏名を取得する
    return name;
  }

  int getAge() {                // メソッド：変数 age から年齢を取得する
    return age;
  }

}
```

先頭の class はこれからクラスを記述することを示すキーワード。 次にクラス名を書く。 クラス名は Java の識別子の命名規則に従ってさえいれば、自由な名前を付けることがでる。 クラス名は大文字で始めるのが慣用。 行末に中括弧始め { を書き、次行からクラスの内容、すなわち変数とメソッドを書く。 最後に行頭の中括弧終わり } で終了する。

コンストラクタ

クラスにはそのクラスと同名のメソッドを書くことができる。 これをコンストラクタと言う。 コンストラクタはそのクラスの生成時に必ず呼び出され、クラスの変数の初期化などを行う。 コンストラクタには戻り値がない。 したがって、戻り値の型も指定しない。 1つのクラスには引数の異なる複数のコンストラクタを定義できる。

コンストラクタの例

以下の例では、3種のコンストラクタを定義している。 クラスにコンストラクタが1つも定義されていないと、コンパイラが自動的に以下の「コンストラクタ (3)」のような、空（から）のコンストラクタを生成する。これをデフォルトコンストラクタと言う。

```
class Person {
  String name = "";
  int age = 0;

  Person(String n, int a) {     // コンストラクタ(1)
    name = n;
    age = a;
  }

  Person(String n) {            // コンストラクタ(2)
    name = n;
  }

  Person() {                    // コンストラクタ(3)
  }
}
```

デフォルトコンストラクタは何もしていないように見えるが、super というメソッドによって上位のクラスを呼び出し、クラス生成の際に必要な準備を整えるという重要な仕事を行う。 これはデフォルトコンストラクタ以外のコンストラクタでも同じ。

コンストラクタの指定方法

それぞれのコンストラクタは次のようにして使われる。 「オーバーロード」のしくみにより、引数の形態に対応するコンストラクタが呼ばれる。

```
      Person("田中", 20)      // コンストラクタ(1)が呼ばれる
      Person("田中")          // コンストラクタ(2)が呼ばれる
      Person()                // コンストラクタ(3)が呼ばれる
×     Person(20)              // 対応するコンストラクタが無いのでエラー
```

インスタンス

クラスは単なる鋳型（設計図）にすぎず、クラスを記述してもそれだけでは利用できない。 使えるようにするにはメモリ領域を確保して、そこに展開し、後で使えるように固有の名前を付ける必要がある。 この作業を一度で指示するのが、これまで何度か出てきた new 文。 new の結果メモリ上に展開されるデータの実体をインスタンスまたはオブジェクトという。

インスタンスの生成

インスタンスの生成は、実際には次のように行う。 まず以下の 01: のように tanaka という変数のデータ型が Person であることを宣言する。 次に 02: のように new 文でメモリ領域を確保しデータを展開する。 new の後ろでは、Person のコンストラクタを呼んでいる。 これは前項のコンストラクタ（1）を呼んでいることになる。

```
01:   Person tanaka;
02:   tanaka = new Person("田中",20);
```

上記 01: は宣言文であり、int age などの型宣言文と同じ構造になっている。 つまり Person は int などと同じようなデータ型の 1 種ともいえる。 Java には基本データ型は 8 種類しかないが、クラスを作ることで、いくらでもデータ型を定義することができる。

まとめた記述

配列と同じように、01: と 02: をまとめて次のように 1 行で書くこともできる。

```
Person tanaka = new Person("田中",20);
```

クラスメンバーの指定方法

インスタンスの生成後、そのメンバー（変数とメソッド）を指定するには、次のように、インスタンス名の後ろにピリオド（ . ）を付けて、その後にメンバー名を記述する。

```
tanaka.age;              // 変数の呼び出し
tanaka.getName();        // メソッドの呼び出し
```

※ 従来のプログラムに慣れた方ですと、なぜ getName(tanaka) ではなくて tanaka.getName() にするのかと思われるかも知れない。 これはもちろんオブジェクトを強調するためだが、この両者に本質的な違いがあるわけではない。 最初は違和感があるかもしれないが、この書き方に慣れること。

アクセス修飾子

前述のようにクラスのメンバーは「インスタンス名.メンバー名」で他のクラスからアクセスすることができる。 しかし、次のようなアクセス修飾子をクラス定義の先頭に付加することで、アクセスの範囲を制限することができる。 アクセス修飾子の種類は以下の通り。 public 修飾子はクラス自身にも付けることができる。 protected と private が付けられるのはクラスメンバーのみ。

(1) public
　　どこからでもアクセスが可能
(2) protected
　　同一パッケージ内のすべてのクラス、および別パッケージのサブクラスからのアクセスが可能
(3) 修飾子なし
　　同一パッケージ内のすべてのクラスからのアクセスが可能
(4) private
　　同一クラスからのみアクセスが可能

※ 「protected（保護された)」の方が、(標準用法と考えられる)「修飾子無し」よりも、保護されていない（すなわちアクセスされる範囲が広い）のは奇妙な事実。 これは protected が、オブジェクト指向言語の初期の段階から命名されていた用語であるのに対し、修飾子無しは Java 言語の設計者が、後から作ったものだから。

一般に変数には private 修飾子を付けて外部からの直接のアクセスを制限する。 そして変数の書き込みメソッドで値をチェックすることで、不正な値が入るのを防ぐことができる。 以下の例では氏名の長さや年齢の値を確認してから書き込んでいる。

```
class Person {
  private String name = "";      // private 修飾子付加
  private int age = 0;           // private 修飾子付加

  void setName(String n) {
    name = n.substring(0,13);    // 長さが 13 以上なら後部をカット
  }                              // substring(a,b)は位置 a から位置(b-1)
                                 // までを切り出すメソッド
  void setAge(int a) {
    if(a<0) { a=0 };             // 値がマイナスなら 0 をセット
    age = a;
  }
}
```

このように変数とそれを処理するメソッドとをまとめて扱うことで、外部からの干渉や誤用を減らすことができる。 この考え方をカプセル化と言う。 カプセル化はオブジェクト指向の重要な概念の一つ。 うまくカプセル化をすると、外部の利用者は内部の構造を知らずに操作が可能であり、逆に内部の開発者は、操作（メソッド）の仕様さえ守れば内部を自由に作成、改変することができる。

final 修飾子

変数、メソッド、クラスに final と言う修飾子を付けることができる。 final 修飾子は変更を許さないということ。 それぞれ次のような意味になる。

変数　　：値を変更できない(つまり定数と同じ扱いになる)
メソッド：オーバーライドできない(オーバーライドは次章参照)
クラス　：継承できない(継承は次章参照)

static メンバー

クラスを使用するには new しなければならないと述べたが、例外がある。 それはメンバーに static と言う修飾子を付ける方法。 static を付加されたメンバーは new しなくてもクラス定義が実行された時点でメモリ上に展開される。 static でないメンバーをインスタンスメンバー（インスタンス変数／インスタンスメソッド）と言う。 1 つのクラスに static メンバーとインスタンスメンバーとを混在して記述することができる。

```
class Person {
  static int total = 0;          // static 変数
  private String name = "";
  private int age = 0;
  以下省略…
```

上の定義が実行されると、static 変数 total が、メモリ上に展開される。 その後 Person を new すると total 以外の部分だけがメモリ上に展開される。 すなわち、インスタンスが複数個 new されても total は 1 つだけになる。 前にも触れましたが、main メソッドも static メンバー。 main メソッドの実行時には、誰もそれを含むクラスを new してくれないので、static にしておく必要がある。

static メンバーのアクセス

インスタンスメンバーは「インスタンス名.メンバー名」と呼び出すが、static メンバーは、インスタンスが生成されないので「クラス名.メンバー名」で呼び出す。

```
tanaka.age;      // インスタンスメンバーはインスタンス名 tanaka を使う
Person.total;    // static メンバーはクラス名 Person を使う
```

this キーワード

クラスの中で自分自身を示したいときには this を使う。 例えば Person クラスの setName メソッドで、引数を n ではなくフルネームで name と書くと、クラスの変数である name と名前が衝突しまう。 このようなときは、次のように this を付けて書くとクラスの変数の name であることを明示でる。

```
String name = "";              // (1)この name はクラスの変数
void setName(String name) {    // (2)この name は引数
  this.name = name;    // this.name は(1)の name、右辺の name は(2)の name
}
```

コンストラクタでの this の使用

コンストラクタ本体の先頭に this（引数）と書くと、引数の型と数に対応した同じクラスの対応するシグニチャのコンストラクタが呼び出さる。 例えばコンストラクタの項で示した例を this を使って書くと次のようになる。

```
Person(String n, int a) {
  this(n);                     // こう書くとすぐ下のコンストラクタが
                               // 呼ばれ name = n; が実行される
  age = a;
}
Person(String n) {
  name = n;
}
```

クラスの配列

1 つのクラスから生成したインスタンスを配列にして扱うことができる。 例えば Person クラスから複数のインスタンスを生成したときは、配列にして管理すると便利。 この配列は次のようにして生成する。

```
Person[] p = new Person[3];
p[0] = new Person("田中", 27);
p[1] = new Person("山田", 20);
p[2] = new Person("佐藤", 34);
```

このように配列の生成と、Person クラスの生成のそれぞれに new を使うところがポイント。 ちょっと複雑だがが、クラスの配列は初心者が最初につまずくところ。 上の例を良くみて理解すること。 クラスを配列にすると、次のようにして容易に全員の氏名を出力することができる。

```
for (int i=0; i<p.length; i++) {
  System.out.println(p[i].getName());
}
```

出力結果　　田中
　　山田
　　佐藤

※ 配列は繰り返しデータを定義できるが、基本データ型の配列では int 型なら全部 int 型となり、同じ要素しか扱えない。 一方 COBOL の OCCURS 句では異なった型のデータをまとめて、それを繰り返すことができた。 クラスの配列を使うと、このように均一でない構造を持ったデータの繰り返しが可能になる。

インスタンス化の必要性

オブジェクト指向言語の初心者は、次のようなプログラムを書いてしまうことがよくある。

```
public class Sample {
  public static void main(String[] args) {
    open(args[0]);      // 前処理
    proc();             // 主処理
    close();            // 後処理
  }
  void open (String[] s) { … }
  …以下略
```

しかしこれでは、open メソッドがメモリ上に展開されていないので、コンパイルエラーになってしまう。 これを避けるには、open メソッドにも static 修飾子を付けて static メソッドにすればよいのですが、こうすると、proc や close にも全部 static を付けることになり、オブジェクト指向プログラムらしからぬものになってしまう。

このような場合には、自分（Sample クラス）自身を new してインスタンス化する。 すなわち、次のようにする。 こうするとオブジェクト指向プログラムっぽくなる。

```
public class Sample {
  public static void main(String[] args) {
    Sample sp = new Sample(); // Sample クラスを new して、インスタンス sp を生成
    sp.open(args[0]);   // 前処理
    sp.proc();          // 主処理
    sp.close();         // 後処理
  }
  void open (String[] s) { … }
  …以下略
```

自分自身を new すると、恐ろしい無限地獄に陥りそうな想定をしてしまうが、そんなことはない。 それは new しているのが static メソッドの中であり、static メソッドはそれを含むクラスを new してもその中には含まれないから。

クラスの構造

オブジェクト指向言語を学習する上で難しいとされる継承。 継承を使うと基準となるクラスに機能を追加した新しいクラスを作ることができる。 継承はカプセル化と並んでオブジェクト指向の重要な概念です。 継承そのものは比較的単純な機能ですが、関連していくつかの概念が出てくる。 これらを包括的に理解することが重要。

継承の例

例を使って継承の働きを説明する。 まず基準となるクラスの例として、おなじみの Person クラスを再度示する。 複雑になるので set メソッドやコンストラクタは省略する。

```
class Person {
  String name = "";             // 変数 氏名
  int age = 0;                  // 変数 年齢

  String getName() {            // メソッド 氏名取得
    return name;
  }

  int getAge() {                // メソッド 年齢取得
    return age;
  }
}
```

継承の方法

上記の Person クラスを継承して Student（学生）クラスを定義してみる。 ここでは学生に特有の変数 grade（学年）とそれを取得するメソッドを追加する。 継承したクラスを作るには以下のようにクラス名の直後にキーワード extends と継承元のクラス名 Person を書く。 中括弧 { } の中に追加するメンバーを定義する。

```
class Student extends Person {  // Person を継承することを示す
  private int grade=0;          // 変数 grade を追加

  int getGrade() {              // メソッド getGrade を追加
    return grade;
  }
}
```

継承の利点

継承の元となるクラス（ここではPerson）をスーパークラス、継承して新しく作るクラス（ここではStudent）をサブクラスと言う。 サブクラスのインスタンスはスーパークラスのメンバーに加え、サブクラスで追加したメンバーを使うことができる。 継承の原語であるinheritance（インヘリタンス）には遺産とか相続の意味がある。 継承とは、まさにスーパークラスの遺産をそっくり受け継いで、最初から自分が持っていたかのように利用すること。

ただそれだけなら、わざわざ継承など使わずに、コピー&ペーストで複写すれば良いのではないかと考えるかも知れない。 しかし、プログラムには修正が付き物。 後から修正しようとしたときにコピー&ペーストで作成した部分を漏れなく探すのは大変。 コードの重複を避けることが保守のしやすいプログラム開発のポイント。

継承したメンバーの使い方

以下ではStudentのインスタンスstがスーパークラスのメンバーとサブクラスのメンバーを使う様子を示す。

```
Student st = new Student();      // Studentクラスのインスタンスstをnew
st.getAge();                     // 元のクラスのメンバーも呼べる
st.getGrade();                   // 新たに定義したメンバーも呼べる
```

※ サブには元のものより劣る語感があるが、サブクラスはスーパークラスより機能が多いことに注意すること。 C++ではスーパー/サブではなく、Base（基本）/Derived（派生）と名付けている。 この名前の方が特徴を表しているかもしれない。 しかしSuper/Subの方が簡単でいい。 親クラス/子クラスと言うこともある。

継承の仕様

継承して作った子（サブ）クラスをさらに継承して孫のクラスを作ることができる。 この連鎖の階層には制限はない。 しかし、通常の業務プログラムでは、むやみに階層の数を増やすのはよくないとされている。

あるクラスは上位のすべてのクラスの（privateでない）メンバーを使うことができる。 また1つのクラスから複数の子クラスを作ることができる。 しかし、逆に1つのクラスが複数の親のクラスを継承することはできる。 すなわち継承の関係はグラフ理論でいう木（tree）構造になっている。

多重継承

1つの子クラスが複数の親クラスを同時に継承することを多重継承と言う。 C++では多重継承ができたが、Javaではプログラムの構造が複雑になりすぎるという理由で、禁止されている。 同等の機能は後に述べるインターフェースを使って実現する。

Object クラス

Java には Object という特別なクラスがある。 このクラスは継承を指定しないクラスに対し暗黙的に継承元になる。 すなわち、以下の 01: のように書いても、02: のように書いたのと同じ。 あるクラスが Object 以外の親クラスを継承しても、その親クラスまたはその上位のクラスがどこかで Object を継承するので、Object はすべてのクラスのスーパークラスとなっている。 Object クラスは木構造の根（root）にあたる。

```
01:   class Person {
02:   class Person extends Object {
```

Object クラスではすべてのクラスに共通のメソッドを提供している。 例えば、各インスタンスの説明を文字列で返す toString メソッドや、後にスレッドの章で説明する wait メソッドなどを提供している。

サブクラスのコンストラクタ

サブクラスはスーパークラスのメンバーを引き継ぐが、コンストラクタだけは自前で作らねばならない。 先に Person を継承して定義した Student クラスのコンストラクタを作ってみる。

```
Student(String n, int a, int g) {       // 氏名,年齢,学年を設定
  super(n, a);           // Person クラスのコンストラクタを呼ぶ
  grade = g;
}
```

最初の super(n,a)はスーパークラス（ここでは Person クラス）のコンストラクタを呼ぶメソッド。 これは name と age をセットしているので、ここでは残りの grade をセットしている。

コンストラクタに関する仕様と注意点

コンストラクタの先頭行に this または super のいずれかがないときは、コンパイラが自動的に super()を付加し、親クラスのコンストラクタを呼ぶ。 したがって、このとき親クラスでデフォルトコンストラクタを定義していないとコンパイルエラーになってしまう。 コンストラクタが 1 つも定義されていないと、自動的にデフォルトコンストラクタが生成されるので問題はないが、デフォルトコンストラクタ以外のコンストラクタを定義したときは、必ずデフォルトコンストラクタも定義するようにすること。

メソッドオーバーライド

メソッドオーバーライドとはスーパークラスで定義したのと同じシグニチャのメソッドをサブクラスで再定義すること。 オーバーロードと名前が似ているのでまちがえないように。

★オーバーロード…同名だが引数が違う:単なる名前の節約
★オーバーライド…引数も含めて同じ:サブクラスでの再定義

オーバーライドの働きを例で示す。 最初に挙げた Person と Student の例を使う。 Person クラスで次のようなメソッド display が定義されていたとする。 これは氏名と年齢を表示するメソッド。

```
void display() {        // 親クラスで定義されている display メソッド
  System.out.print("  氏名 : " + name);
  System.out.print("  年齢 : " + age);
}
```

Student クラスは Person を継承しているので、display メソッドを使うことができる。 しかし、このままではせっかくセットした学年は表示されない。 そこで次のように学年も表示するようにメソッドを定義しなおす。このとき同じ display と言う名前にするのがミソで、これがメソッドオーバーライド。

```
void display() {        // 子クラスでオーパーライドする display メソッド
  super.display();      // 親の display()を呼んで氏名と年齢を表示
  System.out.print("  学年 : " + grade);
}
```

これにより、同じ display メソッドでも、対象が Student クラスのときは学年も表示されるようになる。 このように１つの名前で複数の振る舞いを示すことを、ポリモーフィズム（polymorphism：多態性）と言う。 ポリモーフィズムはカプセル化、継承と並んで、オブジェクト指向の 3 大概念と言う。

オブジェクト指向プログラミングの神髄

少し大げさですが、ここまでがオブジェクト指向プログラミングの神髄といえる。 すなわち呼ぶ側は対象（のある性質）のことを知らずにプログラムを作っているので、対象（のその性質）が変化（つまり仕様変更）しても、プログラムコードに影響が出ない。 また学生以外に、例えば学生クラスの他にビジネスマンクラスを追加して、部署名を表示するようにしたとしても、呼び出し側の変更は不要。

換言すれば、当事者だけが知っていることにより（その仕様に関する）関心の分離（Separation of concerns）が起こり、簡明で変化に強いプログラムを作ることができる。 これこそがオブジェクト指向プログラミングの神髄であり、これによって構築だけでなく保守も容易になる。

メソッドオーバーライドの条件

オーバーライドするには以下の全ての条件を満たしている必要がある。

スーパークラスのメソッドと名称、引数の型／数／順序、戻り値の型が同じ。
スーパークラスのメソッドが final でなくかつ private でない。
アクセス修飾子の範囲がスーパークラスのメソッドと同じかより広い。
throws 文で出す例外の種類がスーパークラスのメソッドと同じか少ない。

スーパークラス変数によるサブクラスの操作

ここでは、オーバーライドによるポリモーフィズムを補完する 2 つの機能について説明する。 第 1 の機能は以下のようにスーパークラスの型の変数にサブクラスのインスタンスを代入することができる。

```
Super s1;        //s1 は Super クラスの変数であることを宣言
s1 = new Sub(); //Sub クラスのインスタンスを生成して s1 に代入
```

第 2 の機能はメソッドがサブクラスでオーバーライドされていると、上記のような変則的な変数 s1 ではオーバーライドされた方のメソッドが動作する。 すなわち s1 のように表面的には Super クラスだが、実体は Sub クラスの場合には、実体側のメソッドを実行する。 これらの機能の効果を例で見てゆく。

まずクラスの配列の例に、Person だけでなく Student も代入する。 以下では山田さんだけが Student で grade（学年）の変数を持っている。

```
Person[] p = new Person[3];
p[0] = new Person("田中", 27);
p[1] = new Student("山田", 20, 3);       // Student クラスを new して代入
p[2] = new Person("佐藤", 34);
```

この配列は Person 型で宣言されていますが（1 行目）、前述の第 1 の機能により、そのサブクラスである Student クラスも要素とすることができる（3 行目）。 さて、ここで先にメソッドオーバーライドの項で定義した display メソッドを以下のようにして呼ぶ。

```
for(int i=0; i<p.length; i++) {
  p[i].display();
  System.out.println("");        // 改行の出力
}
```

すると前述の第 2 の機能により、インスタンスが学生のときだけオーバーライドが働き学年が追加され以下のように出力する。

```
出力結果        氏名：田中　年齢：27
                氏名：山田　年齢：20　学年：3
                氏名：佐藤　年齢：34
```

ここでのポイントは、表示するプログラム（print 文を持つプログラム）は、学年を表示するか否かにまったく関知していない。 これが 2 つの機能の効果。 ポリモーフィズムがうまく活かされて、関心の分離が実現できていることを実感できる。

※ 第 1 の機能の逆である「サブクラスの型の変数にスーパークラスのインスタンスを代入すること」はできない。 キャストして無理に代入すると実行時に ClassCastException 例外が発生する。 この代入を許さない理由はサブクラスのみで定義されているメンバーへのアクセスを実行段階で拒否しなければならないから。

※ この第 2 の機能は dynamic binding（動的束縛）と呼ばれることがある。

オブジェクトコンポジション

継承を覚えるとすぐに使ってみたくなるが、実際には継承がフィットする状況は必ずしも多くはない。 オブジェクトコンポジションを使う方が適切な場合も多い。 オブジェクトコンポジションとは、あるクラス A に別のクラス B を取り込む。 すなわち、クラス A の変数としてクラス B を定義する。 取り込んだクラスに処理をまかせることを委譲と呼ぶ

オブジェクトコンポジションの例を示す Person が携帯電話の契約をしたとする。約を表すクラス Plan を作り、その中に基本料金や無料通話分などの変数と料金を計算するメソッドを定義する。そして Person の中に、Plan クラスを取り込み、p.charge()のようにして、料金計算は Plan クラスに委譲する。 これによって Person クラスでは契約形態やそれに関わる料金計算のことは知らずにすみ、プログラムの作成と保守が容易になると言う。

```
class Person {          // Person クラスの定義
  String name;
  int age;
  Plan p;               // Plan クラスを取り込む
    …
    pay = p.charge();   // Plan クラスに処理を委譲
    …
}

class Plan {            // Plan クラスの定義
  int base;             // 基本料金
  int noCharge;         // 無料通話時間
  public int charge() { // 料金計算メソッド
     …
```

is-a 関係と has-a 関係

サブクラスはスーパークラス（の一種）であると言えるので、継承関係は is-a 関係と呼ばれる。 これに対しオブジェクトコンポジションの関係は has-a 関係または（主述を逆にして）part-of 関係と呼ばれる。 自動車とタイヤとの関係は has-a 関係、自動車と乗り物との関係は is-a 関係。

抽象クラス（アブストラクトクラス）

スーパークラスレベルでは何も書くべきことがないメソッドがある。 例えば Shape（図形）クラスで draw（描画）メソッドがあるとする。 しかし図形という抽象的な概念の図を描画するといっても何をしてよいかわかる。 図形クラスを継承して円のクラスを定義して初めて、中心と半径を取得するなどして描画を実行できる。

このような場合には Shape クラスでは draw メソッドの名前や引数の形式（正確にはシグニチャ）だけを定義して、内容を書かない。 このように内容がないメソッドが抽象メソッド。 抽象メソッドは先頭に abstract キーワードを書き、中括弧 { } は書かずにセミコロンで終わる。 抽象メソッドを１つでも含むとそのクラスは抽象クラスになり、クラス宣言にも abstract を付ける。 以下に抽象クラスの例を示す。

```
abstract class Shape {  // 先頭の abstract が抽象クラスであることを示す
  private int width;
  void setWidth(int w){ // これは非抽象メソッド。内容がある
    width = w;
  }
  abstract void draw(); // これは抽象メソッド。内容がない
}
```

抽象クラスは new できない

抽象クラスは new することはできない。 これを許すと実行できないメソッドを持つインスタンスが存在することになるから。 抽象クラスを継承したクラスを作り、未定義のすべてのメソッドに対し具体的な処理が存在するメソッドをオーバーライドして初めて new できる。 このことをメソッドを実装すると言う。

抽象クラスの意義

わざわざスーパークラスで抽象クラスの draw メソッドなどを書かずに、サブクラスで初めて draw を定義すればよいではないか、と考える方もあるかも知れない。 しかし抽象メソッドとして定義しておくと、サブクラスでそれを実装せずに new しようとするとコンパイルエラーになってしまうので、メソッドの記述（引数の形式も含めて）が強要でき、誤りの少ないプログラムが書ける。 この考えは次項のインターフェースでさらに発展してゆく。

抽象クラスに関する仕様のまとめ

1. 抽象メソッドを１つでも持つクラスは抽象クラスになる。 抽象クラスには abstract キーワードを付けなければならない。

2. 抽象クラスは new することができない。 サブクラスですべての抽象メソッドにコードを実装して、抽象クラスでなくして初めて new できるようになる。

インターフェース

抽象クラスには抽象メソッドが少なくとも 1 つ存在するが、抽象ではない（具体的な処理が記述されている）メソッドも混在しているのが普通。 このような混在を止めて、すべてを抽象メソッドにしてしまったのが、インターフェース。 インターフェースの例を示す。

```
interface Shape {
  int width=3;
  void draw();
}
```

インターフェースを定義するには class の代わりに interface キーワードを使う。 インターフェースは public 属性を持った抽象メソッドと、初期値付きの public かつ static かつ final な変数だけから構成されている。

インターフェースの実装

インターフェースを基にして、実体のあるクラスを作ることをインターフェースを実装すると言う。 インターフェースを実装したクラスの書き方は、継承のときと似ていますが、キーワードは extends ではなく implements を使う。 またクラスの継承と異なり、1 つのクラスで複数のインターフェースを実装することができる。 以下に実装の例を示す。 以下ではクラス Circle は Shape と Serializable の 2 つのインターフェースを実装している。

```
class Circle implements Shape, Serializable {
  void draw(){
      // 円の描画処理
}
```

※ 前述のように Java では複数のクラスの継承（多重継承）は禁じられているので、必要がある場合にはインターフェースを使って同等の機能を実現する。

インターフェースの継承

あるインターフェースを別のインターフェースで継承することができる。 そのときは extends を使う。 インターフェースはクラスと違って複数のインターフェースを継承できる。 インターフェース B と C を継承してインターフェース A を定義するときは次のように書く。

```
interface A extends B, C {
```

インターフェースに関する仕様のまとめ

1. インターフェースに記述できるのは抽象メソッドと初期値付き変数のみである。
2. インターフェースの変数の属性は必ず public かつ static かつ final になる。
3. インターフェースのメソッドの属性は必ず abstract かつ public になる。
4. インターフェースを実装するときは implements を使う。 複数可能である。
5. インターフェースを継承するときは extends を使う。 複数可能である。
6. クラスの継承とインターフェースの実装とを同時に行うこともできる。

仕様書としてのインターフェース（インターフェースの意義）

インターフェースは書式だけを指定したものなので、メソッドの使い方を定義した「仕様書」であると言える。名前と引数構成がよく吟味された抽象クラスがうまく用意されていれば、プログラマーは要求されたメソッドを記述していくだけで望ましいクラス構成が実現できる。 初めにインターフェースをきちんと設計しておき、それを実装する形でプログラムを書くのが優れた開発方法であると言われている。

instanceof 演算子

継承や実装を行うと、あるインスタンスが複数のクラスやインターフェースに属していることになる。 インスタンス x がクラス A に属しているか否かは以下のようにして調べることができる。

```
if(x instanceof A)
```

A と B をインターフェース、X と Y と Z をクラスとする。 X が Y を継承し、かつ Y が Z を継承していて、さらに X が A と B を実装しているとき、X のインスタンス x は ABXYZ のすべてに属している。 また Object クラスはすべてのクラスのスーパークラスですので、x は Object にも属している。

[END]