第6回：継承，多態性

継承 (インヘリタンス，inheritance)

継承はオブジェクト指向の重要な概念の1つであり，構造化言語には存在しないものである．

継承とは

あるクラスAがあった時，Aの機能はそのままに，少しだけ拡張した別のクラスBを作りたいときがあるとする．このような時すぐに思いつく方法は，Aのコードをそのままコピーしてペーストし，追加部分をBに書き足すことである．一見，問題なさそうだが，同じコードがクラスAとクラスBの2か所に重複してしまう（**コードクローン**という）ので，次の問題が生じる．

- クラスAを修正したらクラスBも修正しないといけなくなる．
  →よく忘れるし，すべての重複個所を覚えておくことは無理．
- コード量が増えて読みにくくなり，保守しづらくなる．

このような時，**継承**を使うことで，クラスAの属性・操作をそのまま引き継いだ新しいクラスBを作ることができる．クラスBには，Aから拡張した差分のみを書くだけでよく，コードクローンの問題も起こらない．

継承の仕組み

例を用いて継承の仕組みを見てみよう．第4回で登場した学生クラスを思い出してほしい．学生(Student)クラスは，属性（フィールド）として学籍番号，名前，単位数を持っていた．また，操作（メソッド）として，自己紹介，卒業報告，単位追加を行うことが出来た．

いま，新しく留学生(InternationalStudent)クラスを考える．**留学生は学生である** (InternationalStudent is a Student)ので，学生クラスが出来ることはすべてで出来なくてはならない．さらに留学生クラスでは，属性として出身国と国費／私費の区分を持ち，操作として自分がどんな留学生かを説明出来るとしよう．

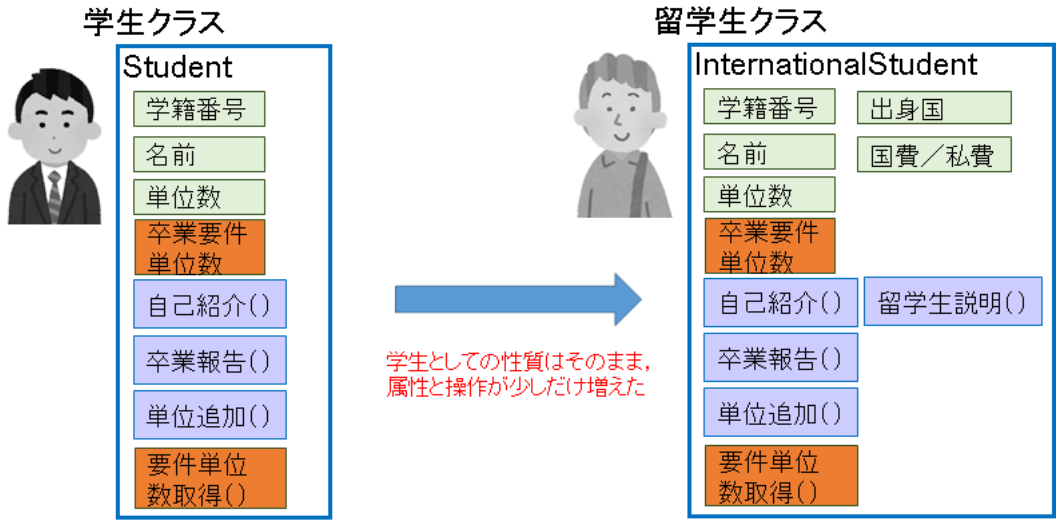

図１：学生クラスと留学生クラス

図1にその概念図を示す．留学生クラスは学生クラスの全属性・操作を持ちつつも，新しく2つの属性と1つの操作が増えている．留学生クラスを作るとき，学生クラスのコードをそのままコピーすることも可能だが，自己紹介のやり方や卒業要件数が変更された時など，2か所のコードを書き換えなくてはならなくなる．

このような場合，留学生を学生を継承する形で定義する．下の図２を見てみよう．

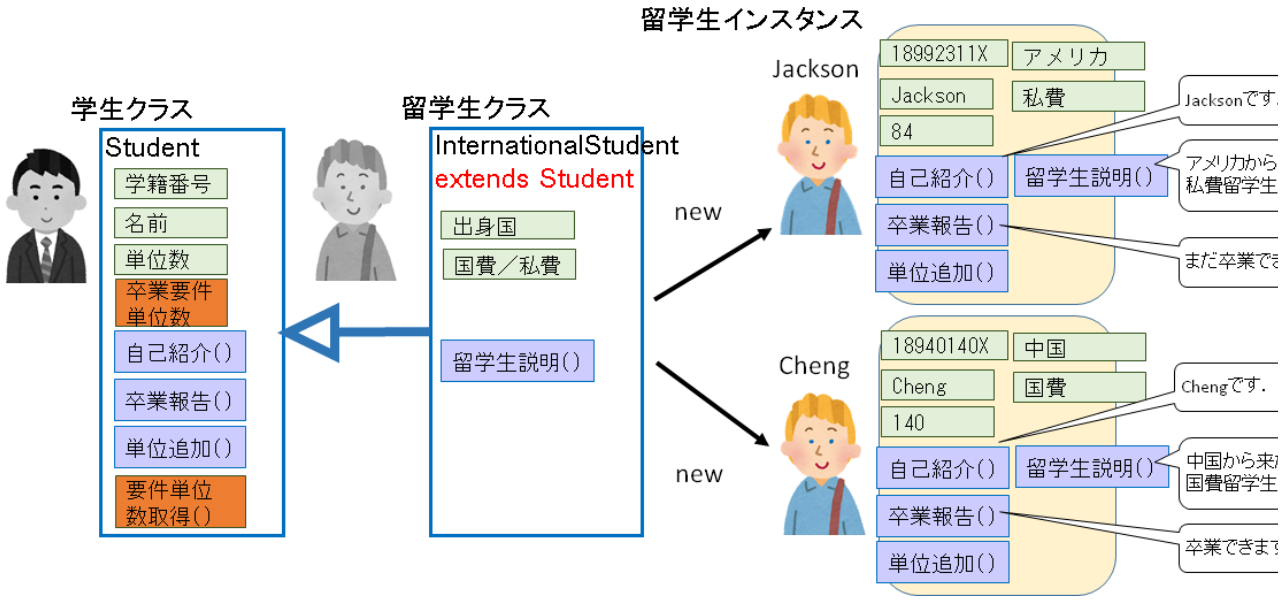

図２： 学生クラスを継承して留学生クラスを定義する

図2は，留学生クラスを学生クラスを継承して定義した概念図である．InternationalStudent extends Studentは，留学生クラスが学生クラスを継承したものであるという宣言である．クラス図的には，留学生から学生に向けて，白抜きの三角矢印が結ばれることになる（**矢印の向きに注意しよう**）．この時，学生クラスを**親クラス**，留学生クラスを**子クラス**（あるいは**サブクラス**）と呼ぶ．

継承が行われたため，子クラスである留学生クラスは親クラスの学生クラスが持つすべての属性・操作をそのまま引継ぐ（わざわざコードに書かないてよい）．留学生クラスに必要な拡張部分のみを新たに追加するだけで，図1と同じ留学生クラスになるのである．

留学生クラスからnewによって生成された留学生インスタンスは，学生クラスの属性・操作はそのままに，出身国および国費私費区分に関する新たな振る舞いが実現される．つまり，留学生は学生の性質を引き継いだ（継承した）まま，特別な機能が追加されたものになる．

ソースコード：InternationalStudent.java

新たにJavaプロジェクト SoftEng06を作成しなさい．その中に新しくクラスStudentを作成し，第5回のカプセル化後のStudent.javaのコードをコピーして貼り付けなさい．

次に，新しくクラスInternationalStudentとMain.javaを作成し，以下のコードをそれぞれ貼り付けなさい．

最後に，Main.javaを実行し，実行結果を確認しなさい．

InternationalStudent.java

```
/*
 * 留学生クラス．学生クラスを継承している．
 */
public class InternationalStudent extends Student{

	/* 留学生用のフィールド */
	private String country;  //出身国
	private boolean kokuhi;  //国費かどうか

	/*
	 * デフォルトコンストラクタ．空の留学生を作成する．
	 */
	public InternationalStudent() {
		//コンストラクタでは，必ず最初に親クラスのコンストラクタを
		//呼び出してから，自前の前処理を書く．

		//↓super()は親クラスのデフォルト・コンストラクタを呼び出すという意味．
		super();

		//後は，特に何もしていない．
	}

	/*
	 * 引数付きコンストラクタ．
	 * Studentのコンストラクタを呼び出して，値をセットしている．
```

```
         */
        public InternationalStudent(String id, String name, int credit,
                        String country, boolean kokuhi) {

                //↓super(....) は親クラスの引数付きコンストラクタを呼び出すという意味.
                //Studentの引数付きコンストラクタに学版, 名前, 単位数を渡す.
                super(id, name, credit);

                //その後, 留学生用のフィールドに出身国, 国費・私費区分をセット.
                this.setCountry(country); //thisは省略可能
                this.setKokuhi(kokuhi);   //thisは省略可能
        }


        /* (4) 留学生であること（出身国と国費・私費区分）を説明する.  */
        public void explain() {
                System.out.print("私" + getName() + "は, " + country + "から来た");
                /*↑getName()は, 親クラスのメソッド(nameのgetter)を呼び出している.
                 * super.getName()と書くこともできる.
                 * ※親クラスの属性 nameはprivateなので直接参照できない */

                if (kokuhi==true) {
                        System.out.println("国費留学生です. ");
                } else {
                        System.out.println("私費留学生です. ");
                }
        }

        /* ----------------- 留学生用のsetter/getter ----------------------*/
        public String getCountry() {
                return country;
        }
        public boolean isKokuhi() {
                return kokuhi;
        }
        //↓setter. 厳密には値チェックをしたいところ
        public void setCountry(String country) {
                this.country = country;
        }
        public void setKokuhi(boolean kokuhi) {
                this.kokuhi = kokuhi;
        }

}
```

Main.java

```
public class Main {

        public static void main(String[] args) {
                //3人の学生インスタンスを生成
                Student s1 = new Student("12345678X", "中村", 136);
                Student s2 = new Student("18512245X", "前田", 86);
                Student s3 = new Student("18416623X", "中谷", 128);

                //2人の留学生インスタンスを生成
                InternationalStudent is1 =
                        new InternationalStudent("18992311X", "Jackson", 84, "アメリカ", false);
                InternationalStudent is2 =
                        new InternationalStudent("18940140X", "Cheng", 140, "中国", true);

                /* あいさつ（全員共通）*/
                s1.hello();
                s2.hello();
                s3.hello();
                is1.hello();
                is2.hello();
                System.out.println("----------------------");

                /* 留学生であることの説明（留学生だけできる）*/
                is1.explain();
                is2.explain();
                System.out.println("----------------------");


                /* 卒業確認（全員共通)*/
                s1.graduate();
```

```
                s2.graduate();
                s3.graduate();
                is1.graduate();
                is2.graduate();
                System.out.println("----------------------");

                /* 全員に50単位追加 (全員共通)*/
                s1.addCredit(50);
                s2.addCredit(50);
                s3.addCredit(50);
                is1.addCredit(50);
                is2.addCredit(50);
                System.out.println("----------------------");

                /* 再び卒業確認 (全員共通)*/
                s1.graduate();
                s2.graduate();
                s3.graduate();
                is1.graduate();
                is2.graduate();
                System.out.println("----------------------");

                /* Student型のリストを作成し，5人全員を入れる */
                ArrayList<Student> list =  new ArrayList<Student>();
                list.add(s1);
                list.add(s2);
                list.add(s3);
                list.add(is1); //留学生も学生なので，そのまま入れられる！
                list.add(is2); //留学生も学生なので，そのまま入れられる！

                // まとめて表示してみる
                for (Student s: list) {
                        System.out.println(s); //toString()が呼ばれて文字列で出力される
                }

                System.out.println("----------------------");
                // あいさつもまとめてできるが，留学生の説明はできない．．．
                for (Student s: list) {
                        s.hello();
                        /*↓sは学生とみなされているので，留学生の振る舞いはできない*/
                        // s.explain();
                }

        }
}
```

解説

- クラス宣言の extends Student は，このクラスがStudentクラスを継承することの宣言．
- コードには陽に書いていないが，StudentクラスのメンバがinternationalStudentクラスですべて引き継がれる．
- 継承したクラスでのコンストラクタでは，まず親クラスのコンストラクタを呼び出す決まりがある．親クラスのコンストラクタは，以下のように書く．
  - super(); //デフォルトコンストラクタを呼び出す
  - super(引数リスト); //引数付きコンストラクタを呼び出す
  これらは省略できるが，暗黙的にはsuper();が勝手に挿入されることになる．
- 引数付きコンストラクタは，学籍番号，名前，単位数，出身国，国費かどうかの5つの引数を受け取り，留学生インスタンスを作成する．その際，前3つの引数を親クラスの学生クラスのコンストラクタに渡して，学生クラスが持つ3つの属性に代入，その後，後2つの引数を留学生クラスのsetterにそれぞれ渡している．
- explain()メソッドでは，親クラスが持つフィールドnameにアクセスしようとするが，親クラスでprivateとしてカプセル化されてしまっているため，getterで取り出している．
- Main.javaでの振る舞いを丁寧に追いかけ，出力結果を理解しよう．
  1. まず 3人の学生インスタンス(中村，前田，中谷)を作成．
  2. 次に2人の留学生インスタンス（Jackson, Cheng）を作成．
  3. 全員あいさつhello()を実行する．留学生も学生であるから同じようにあいさつができる．
  4. 留学生2名が留学生であることの説明explain()を実行している．これは普通の学生はできない操作．
  5. 全員が卒業確認を行う．これは全員共通．
  6. 全員に50単位を追加．これも全員共通．
  7. 再び卒業確認．これも全員共通．

8. 最後のブロック．学生を入れるArrayListを作成し，その中に5人全員を入れている．留学生は学生を継承しているので，学生とみなされリストに格納可能である！

- クラス図は以下のようになる．子クラスから親クラスへ白抜きの三角矢印を描く．

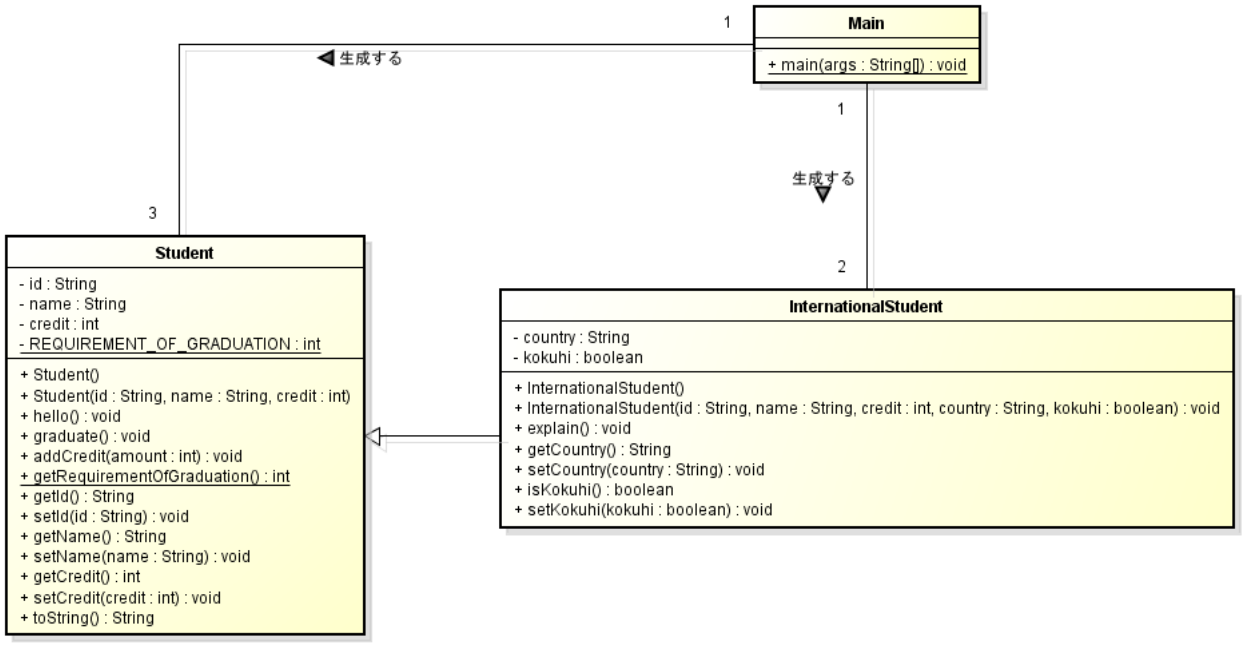

図３： 例題のクラス図

オーバーライド (Override)

上のMain.javaの最後で，留学生も学生とみなしてリストに入れることで，まとめていろんな処理ができることが分かった．これは継承によって「留学生 is a 学生」という構造が宣言されたためであり，これを利用することで非常に効率的なコードが書けるようになる．

一方で，留学生を学生とみなしてしまうことで不便なこともある．Main.javaの最後の拡張for文内で，学生があいさつhello()する際に，留学生の説明explain()が実行できなくなってしまう（「sが留学生の時にはexplainを実行する」というコードを書けなくもないが見栄えも悪いし，煩雑になる).

それなら，いっそのこと留学生については，あいさつの仕方hello()を書き換えて，「留学生特有のあいさつ」をしてはどうかという考えが生まれる．下記のメソッドのコードをInternationalStudent.javaのexplain()メソッドの下あたりに，挿入してみよう．

InternationalStudent.javaへの追加コード

```
	/* (5) 留学生のあいさつの仕方 */
	@Override
	public void hello() {
		super.hello(); //普通の学生のあいさつの後に,
		this.explain(); //留学生の説明をする.
	}
```

その後，Main.javaを実行するとどうなるだろうか？最後の拡張for文で，留学生の場合はその説明が出るようになったと思う．

hello()の2重定義では！？

図２と 図３を見ながら考えてみよう．今やったことは，留学生クラスに自己紹介()すなわちhello()を新たに付け加えたことになる．親クラスのStudentにも同じ名前で同じ引数のメソッドhello()が存在するから，図1の考えに立ち返れば，これはhello()の2重定義になるのではないか？関数の重複定義でコンパイルエラーになる？

メソッドのオーバーライド

実は，継承した子クラスで親クラスと全く同じメソッド（名前も引数の型も同じ）を定義することは許されている．これをメソッドの**オーバーライド（Override）**と呼び，子クラスのメソッドが親クラスのメソッドを上書きしたものと解釈される．つまり，留学生インスタンスに対してhello()が呼ばれた時には，留学生クラスで定義されたhello()が優先して呼び出される．

@Override： オーバーライドの印

メソッド宣言の上に書かれた@Overrideは，**Javaのアノテーション**という機構の一つで，親クラスのメソッドをきちんとオーバーライドできているかをコンパイラがチェックしてくれるものである．つけておくとよい．

フィールドのオーバーライド

フィールドはオーバーライドできない．

静的メソッドのオーバーライド

静的メソッドはクラスに紐づくため，オーバーライドできない（というかする必要がない．いつでも親クラス名.メソッド（）で呼び出せるので）．

以前どこかで見たことが．．．

第5回のカプセル化後のStudent.javaのコードのtoString()メソッドを見てみよう．@Overrideがついていることが分かる．実はJavaの全てのクラスは，暗黙的にObjectというクラスを継承している．Objectクラスは，Javaにおける全てのオブジェクトのひな形を定義するクラスで，インスタンスの文字列表現を行うtoString()を含むいくつかのメソッドが定義されている．Studentクラスでは，学生用のtoString()を自前で定義したが，これはObjectクラスのtoString()を上書き（オーバーライド）して定義したことに他ならない．これにより，println()でインスタンスを文字列として出力できるようになる（なぜかは多態性のところで説明する）．

多態性（ポリモーフィズム，polymorphism）

継承とオーバーライドがもたらすメリットを，先ほどのコードに基づいて考えてみよう．

呼び出す側 (Mainクラス)

留学生も普通の学生も同じ学生として扱うことができ，同じ学生リストに入れられる．
留学生にも学生と同じ操作を指示できる．

呼び出される側 (留学生クラス，学生クラス)

指示された操作が自分のクラスで定義されていなければ，親クラスの操作を探して実行する．
指示された操作が自分のクラスで定義されていれば，それを実行する．
指示された操作が自分のクラスと親クラスの両方に定義されていれば，自分のクラスの操作を優先する（オーバーライドの原理）．

このことを図示すると，図４のようになる．

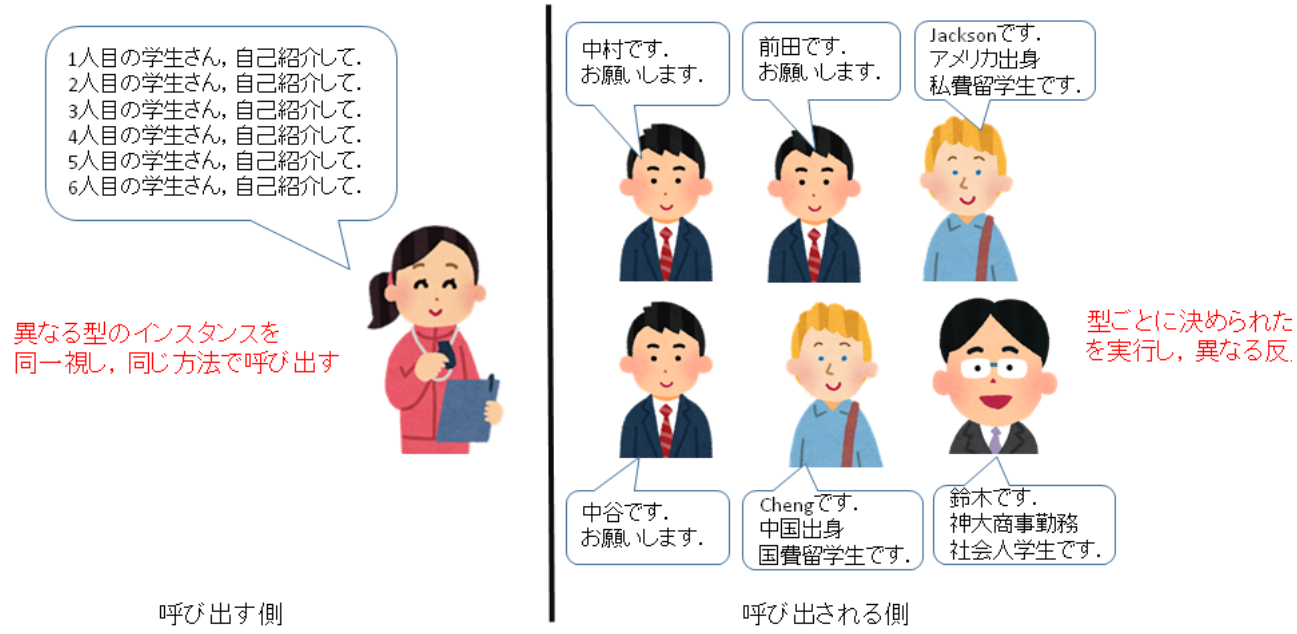

図４： 多態性（ポリモーフィズム）

対応するコードはこのようになる．

```
/* ポリモーフィズム */
//呼び出す側は学生も留学生も両方学生とみなして，
Student s = new Student("12345678X", "中村", 136);
```

```
        Student is = new InternationalStudent("18992311X", "Jackson", 84, "アメリカ", false);

        //同じように学生として自己紹介を指示できる.
        s.hello();
        is.hello();

        /* ↑の結果，呼ばれた側は，学生と留学生それぞれ異なる適切な自己紹介ができる.
         *（学生）　中村です．よろしくお願いします.
         *（留学生）Jacksonです．アメリカから来た私費留学生です
         */
```

同じ呼び出し方なのに異なる反応が返る

左側の先生（Main.java）は，右側にいる6人のインスタンス（学生，留学生，社会人学生）をすべて学生だとみなしている．すべて学生なので，6人に対して同一の操作を指示している．一方，右側のインスタンスは，3人が学生インスタンス，2人が留学生インスタンス，1人が社会人学生インスタンスであり，留学生と社会人学生はともに学生クラスを継承し，自己紹介メソッドhello()をオーバーライドしているとしよう．先生からの指示に従って，それぞれのインスタンスは，それぞれのクラスで決められたやり方で自己紹介をしている．その結果，インスタンスの型ごとに異なる自己紹介の反応が得られている．

多態性（ポリモーフィズム，polymorphism）

このように，同じ呼び出し方をしたにも関わらず，インスタンス毎に適切な挙動を生み出せるという性質を，**多態性（ポリモーフィズム）**と呼ぶ．多態性によって，呼び出し側は意識しなくても，実行時の条件に合った適切なメソッドが実行されるという，巧妙な仕組みを実現できる．多態性は，カプセル化，継承と並んでオブジェクト指向の重要な概念の一つである．

多態性は機能と実装を分離する

より専門的には，呼び出し側の指示（メッセージの送信）に対する呼び出される側の反応（メッセージの受信）を実行時に決める（動的バインディング）ことで実現され，機能と実装の分離を実現する．自己紹介するという機能（What）と，それをどうやって実現するかという実装(How)を分けて考えることができるのである．人間同士のコミュニケーションでは普通に行っていることだが，プログラム言語では複雑な機構で実現される．

多態性と単態性

従来のプログラム言語では，プログラムの要素はただ1つの型に属すると考え方（単態性）に基づいていた．一方，多態性はこうした要素が複数の型に属すことを許す性質である．呼び出し側から見えるStudent.hello()は，実際にはStudent, InternationalStudent, WorkingStudentの3つのクラスのどれかに属したhello()が実行されることになる．

```
/* オーバーライド，多態性の仕組みがないとこんなコードになる */

Student s = ....; //学生か留学生かどちらか.

if (s instanceof InternationalStudent) { //sが留学生インスタンスの時
        s.hello(); //学生あいさつ
        ((InternationalStudent) s).explain(); //キャストしてから留学生説明
} else { //sが学生インスタンスの時
        s.hello(); //学生あいさつ
}
```

toString()のオーバーライドとprintln()の関係

学生クラスにpublic String toString()を追加すると，System.out.println()で表示できるようになる．これはまさに多態性を利用した仕組みである．println()は引数にObject型を取ることができ，Object型のインスタンスが来た時にはその文字列表現を得て，それを標準出力に書き出す．あらゆるクラスはObject型を継承しているので，学生インスタンスをそのままprintln()に渡すことができる．println()は中でObjectの文字列表現を得ようとし，Object.toString()を呼び出す．学生クラスには自前のtoString()をオーバーライド定義したので，それが呼び出されて定義した文字列表現が出力されるのである．

継承で注意すること

継承はオブジェクト指向の花形と言われ，使ってみたい気持ちに駆られる．一方で，本質をよく理解せずに継承を乱発すると，とんでもなく複雑なプログラムができてしまう．

継承のデメリット

継承は親クラスの操作や属性を引き継ぐことで，少ないコードの量で容易に拡張クラスを作成できるメリットがある．さらに，上で述べた多態性というスマートなオブジェクトの扱いは，継承なくしては実現できない．しかしながら一方で，継承はクラス間の依

存を強めてしまうため，プログラムを保守しにくくなるというデメリットがある．

親クラスの変更が子クラスに波及する

あるクラスAが100個のサブクラスを持っていた場合，Aを変更すると100個のサブクラスに波及する．変更後，100個のサブクラスが正常に動くとは限らず，テスト・動作確認をしなければならなくなる． こうなると怖くてAを変更できなくなる．

継承が深くなると訳が分からなくなる

継承を何段も使って差分プログラミングを行っていくと，もともと何のクラスから始まったのかわからなくなってくる．学生クラスから継承を深めていった結果，気が付けば事務員クラスが出来ていたということになりかねない．

継承の使いどころ

継承を使うべきか使わない方が良いかを決める目安をいくつか挙げておこう．

本当に is-a の関係にあるか？

クラスBをクラスAを継承して定義するとき，「B is a Aの関係にあるか」を厳しくチェックすべきである．単に機能を再利用したいからという理由で，安易に継承すべきではない．親子の関係は一度定義すると死んでも切れないのである．
- 良い例： 留学生 is a 学生， 社会人学生 is a 学生，プレミアム会員 is a 会員
- 悪い例： 社会人学生 is a 留学生，留学生 is a 友達， 友達リスト is a 受診者リスト

本当に必要なのか？

単に「使ってみたい！」「かっこいい！」という理由で，無理やり人工的な継承を定義していないか？技術に走った結果，もともと解決したい問題に必要のないクラスが濫立してしまうことがよく起こる．
- 本当に必要？： 学生 is a 人間，人間 is a 哺乳類，

多態性の効果が発揮できるか？

継承を使うことで，多態性のメリットが受けられているか？メリットが薄い場合には，わざわざ使わなくてもよい

Javaでは多重継承はできない

「社会人学生は学生だけれど，一方では社会人だし．．．」ということから，public class 社会人学生 extends 学生，社会人としたいことがあるかもしれない．このように，複数の親クラスを継承することを**多重継承**と呼ぶ．Javaでは多重継承が許されていない．概念上の話では多重継承はありうるが，実際のシステム開発においては本当に必要な場面は稀である．学生管理システムを作っているときに，（学生ではない）社会人クラスは本当に必要なのか？
- 本当に必要？： 加湿空気清浄機 is a 加湿器 and a 空気清浄機

世の中の流れ的には

「継承を使わずに済むところは使わないで済まそう」「継承より委譲を使おう」「継承はis-aのモデリングがしっくりいく場合のみ使う」という流れがある．継承がもたらすクラス間の密結合を避けるため，不用意な継承は慎むべきという警鐘である．

演習問題

- 第6回演習問題へ

masa-n@cs.kobe-u.ac.jp

Last updated: 12/21/2020 17:26:30