第5回：コーディング規約，カプセル化，クラス図

コーディング規約

Javaプログラマの世界では，ソースコードの見やすさ，保守のしやすさを目的として，コードの書き方に関する多くの「作法」が定められている．いわゆる，コーディング規約やコーディング標準と呼ばれるものである．

これらの作法は，守らなくてもエラーにはならないが，世の中のプログラマが踏襲している標準には従いたいものである．本講義では，特に重要なものを抜粋して紹介する．

命名規則（名前の付け方）

変数や関数，クラスに名前を付ける際の決まりごと．

スペル ：

英数文字を使った単語を使う．名前が複数の単語に分かれる場合には，キャメルケース を使う．記号は使わないこと

キャメルケース (camel case) ：

複合語のスペースを詰めて，次の語を大文字にする書き方．ラクダのこぶのように見えることから名付けられた．

スネークケース (snake case)：

複合語をアンダーバーでつなげる書き方．C言語等で使われる書き方．

表 1 ： キャメルケースの書き方サンプル

元の複合語	キャメルケース (Java)	スネークケース(C)
call name	callName	call_name
get score	getScore	get_score
Buffered reader	BufferedReader	Buffered_reader
show fibonacci series	showFibonacciSeries	show_fibonaci_series
XML HTTP request	XmlHttpRequest	XML_HTTP_request

クラス名 ：

大文字で始まる名詞でつけること．可能な限り意味のある名詞をつけよう（クラス名は非常に大事）．

- 良い例： Car, Product, SuperCar, BufferedReader
- 悪い例： id(小文字で始まる), MYCLASS(区切り不明), PrintProductList(動詞形), XYZ(意味不明), Sample1(漠然としすぎ), Kadai2_1(気持ちはわかるが．．．)

フィールド変数名 ：

小文字で始まる名詞でつけること．フィールドはそのクラスの属性であり，クラス内の全メソッドから参照できるため，わかりやすい名詞をつけよう．

- 良い例： price, book, name, age, customerId
- 悪い例： Price(大文字で始まる), reserve(動詞)

ローカル変数名：

小文字で始まる名前．メソッド内で一時的に使うため，厳密に名詞になっている必要はないが，分かりやすさ重視でつけよう（クラスの型の頭文字，省略系が使われる事が多い）．

メソッド名：

小文字で始まる動詞でつけること．メソッドはそのクラスの振る舞いを表すものであるから，意味のある動詞＋目的語（または動詞単独）でつける．

- 良い例： show(), getName(), setCustomerId(), sendAllProducts()
- 悪い例： SwitchOn() (大文字で始まる), mainMenu() (名詞形) ,calc() (舌足らず)

定数名：

定数 (static final) は，すべて大文字にする．この場合スネークケースにする．

- static final int REQUIREMENT_OF_GRADUATION = 130; // 卒業要件単位数

パッケージ名 (次回以降)：

全て小文字の名前をピリオドでつなげてつける．

- newse.lesson.example
- jp.kobe_u.cs.nakamura

インデント

インデント（字下げ）はコードの読みやすさに大きくかかわる．Eclipseの自動インデント機能 Ctrl+Shift+F を習慣にしよう．

コメント

ソースコードにはなるべくコメントを入れる．他の人に公開する場合には，必須である．

```
// 1行コメント

/* コメント行1
 コメント行2 */

/*
 *  よりスタイリッシュなコメント
 *
 */

/**
 *   Javadoc(後述)用のコメント
 *
 */

/**  Javadoc用の1行コメント */
```

Javaのオリジナルのコーディング規約

Code Conventions for the Java Programming Lanuguage

カプセル化

カプセル化 (encapsulation) とは

インスタンスやクラスのメンバー（フィールド，メソッド，コンストラクタ）に **外から直接アクセスできないように制限・隠ぺいする**こと．カプセル化によって，メンバに対して，「外部公開してよい」「内部だけアクセスを許す」「同じパッケージ内だけアクセスを許す」といった細かい**アクセス制御**が可能になり，より堅牢なプログラムが実現できるとされている．

カプセル化はオブジェクト指向を構成する主要な概念の一つである．オブジェクトをカプセルに入れて守るというニュアンスがある．「外に見せなくていいものは見せない」というポリシーで設定する．

メンバの可視性

カプセル化を実現するために，Javaではクラス内の各メンバに対して，以下の4種類の可視性(Visibility)を定義できるようになっている．

表1: Javaの4種類の可視性

可視性	読み方	意味
private	プライベート	自分自身のクラスのみからアクセス可能
(何も書かない)	パッケージ・プライベート	自分と同じパッケージに属するクラスからアクセス可能
protected	プロテクテッド	自分と同じパッケージに属するか，または，自分を継承したサブクラスからアクセス可能
public	パブリック	全てのクラスからアクセス可能

これまで，全てのクラスやメソッド，コンストラクタにpublicをつけてきたが，これは自分以外のクラスにアクセスを公開するための宣言だったのである．

カプセル化の指針

- メソッドは基本的にpublicにする．そのクラスの中だけでしか使い道が無いようなものはprivateにしてもよい．
- コンストラクタも基本的にはpublicにする．new出来ないようにあえてprivateにするデザインパターン(Singletonパターン)も存在するが，発展的内容である．
- フィールドはprivateにし，外から直接アクセスできないようにする．値の取得や代入は，フィールドにアクセスするための専用のpublic メソッド(アクセサという)を通してのみ行う．

なぜカプセル化が必要なのか？

例として，第4回で出てきたStudentクラスを見てみよう．

Student.java (カプセル化前)

```
/**
 * 学生クラス．学番，名前，単位数を持つ
 */
public class Student {
        /*------ 学生の状態を決めるフィールド群 -----*/
        String id; // 学籍番号
        String name; // 名前
        int credit; // 単位数

        /* 静的フィールド．全インスタンスで共通 */
```

```
        static final int REQUIREMENT_OF_GRADUATION = 130; // 卒業要件単位数

        /**
         * デフォルトコンストラクタ．空の学生インスタンスを生成する
         */
        public Student() {

        }

        /**
         * コンストラクタ．学番，名前，単位数を指定して，学生インスタンスを生成する．
         */
        public Student(String id, String name, int credit) {
                this.id = id;
                this.name = name;
                this.credit = credit;
        }

        /*------------------ 主要な学生の振る舞い ---------------------*/
        /** (1) 自己紹介をする */
        public void hello() {
                System.out.println("こんにちは．学籍番号" + id + "の" + name + "です．"
                                + "よろしくお願いします．");
        }

        /** (2) 卒業報告をする */
        public void graduate() {
                System.out.print(name + "の単位数は，" + credit + "です．");
                if (credit >= REQUIREMENT_OF_GRADUATION) {
                        System.out.println("卒業要件を超えているので，卒業できます．");
                } else {
                        System.out.println("卒業要件に足りないので，卒業できません．");
                }
        }

        /** (3) 単位数を追加する */
        public void addCredit(int amount) {
                System.out.println(name + "の単位数を " + amount + "追加します．");
                credit = credit + amount;
        }


        /** 卒業要件数を取得する (静的メソッド)*/
        public static int getRequirement() {
                return REQUIREMENT_OF_GRADUATION;
        }

}
```

上のコードでは，メソッドとコンストラクタにはpublicがついており，カプセル化の指針には沿っている．しかしながら，フィールドには何の修飾子もついていないため，表1によりパッケージ・プライベートになる．よって，同じパッケージ内（デフォルトパッケージ）に存在するクラス，たとえば，Main.javaから以下のようにアクセスできる．

Main.java

```
public class Main {

        public static void main(String[] args) {

                // 空の学生インスタンスを生成する．
                Student s = new Student();

                // フィールドに値を直接代入
                s.id = "1234567X";
                s.name = "中村";
                s.credit = 136;

                // フィールドの値を直接読み出し
                System.out.printf("学番:%s, 名前：%s, 単位数：%s\n",
                                s.id, s.name, s.credit);
```

```
        }
}
```

一見，何の問題もなさそうであるが，例えば次のようなこともできてしまう．

```
public class Main {

        public static void main(String[] args) {

                // 空の学生インスタンスを生成する．
                Student s = new Student();

                // フィールドに値を直接代入
                s.id = "";
                s.name = "";
                s.credit = -136;

                s.hello();

        }
}
```

学生インスタンスsに対して，学籍番号，名前ともに空文字列，負の単位数を代入して，学生インスタンスとして意味的に成り立たないものが出来上がっている．フィールドを外にオープンにしている時点で，このような操作を防ぐことができない．

アクセサ (accessor)

フィールドをprivateにすると，外から値の代入や取得（参照）ができなくなってしまう．その代わりに，フィールドにアクセスするためのpublicなメソッド（＝アクセサ）を用意する．

アクセサはフィールドの値を代入(セット)，取得(ゲット)する対のメソッド

フィールドに対して，値の代入を行うアクセサをsetter (セッター)，値の取得を行うアクセサをgetter (ゲッター)と呼ぶ．各フィールドに対してsetter/getterの対のアクセサを用意することが一般的である．

アクセサの名前の付け方

コーディング規約により，**アクセサの名前の付け方は以下のように決まっている**．覚えてしまおう．

フィールド名がvarでその型がtypeの時

- public void setVar(type var) { } //varの値を設定するメソッド．varのsetterという．
- public type getVar() { } //varの値を返す（取得する）メソッド．varのgetterという．
- public boolean isVar() { } //varがboolean型の場合のgetter．

Eclipseの便利な機能 (アクセサの自動生成)

「ソース」→「GetterおよびSetterの生成」でアクセサを生成したいフィールドを選択する．getter/setterのコードのひな形が自動的に挿入される．

メソッドtoString() の作成

フィールドをprivateにすると，外から直接値を見ることができなくなる．インスタンスの状態を確認するためには，多くのgetterを書く必要が出てきて，確認用のコードが煩雑になる．

このような場合，public String toString()というメソッドを作成すると便利である．toString()は，オブジェクトを文字列表現で出力するという意味のメソッドであり，このメソッドがあると，オブジェクトをSystem.out.println()等で表示することが可能になる．

public String toString() の作り方

フィールドの値を文字列でつなげて文字列で返すような実装にする．書式は任意．確認しやすいようにする．

Eclipseの便利な機能 (toString()の自動生成)

「ソース」→「toString()生成」で表示したいフィールドを選択する．「コードスタイル」から文字列連結やformatなどいろんなtoStringの書き方を選べる．OKを押すと，toString()のコードのひな形が自動的に挿入される． 必要に応じて整形する．

カプセル化後のコード

上記のStudentクラスにカプセル化を適用したコードを見てみよう．

Student.java (カプセル化適用後)

```
/**
 * 学生クラス．学番，名前，単位数を持つ．カプセル化後
 */
public class Student {
	/*------ 学生の状態を決めるフィールド群 -----*/
	//フィールドはprivateにする．
	private String id; // 学籍番号
	private String name; // 名前
	private int credit; // 単位数

	/* 静的フィールド．全インスタンスで共通 */
	private static final int REQUIREMENT_OF_GRADUATION = 130; // 卒業要件単位数

	/**
	 * デフォルトコンストラクタ．空の学生オブジェクトを作る
	 */
	public Student() {

	}

	/**
	 * コンストラクタ．学番，名前，単位数を指定して，学生インスタンスを生成する．
	 */
	public Student(String id, String name, int credit) {
		//無効な値に備えて，setterを呼び出す．
		this.setId(id);
		this.setName(name);
		this.setCredit(credit);
	}

	/*------------------ 主要な学生の振る舞い ---------------------*/
	/** (1) 自己紹介をする */
	public void hello() {
		System.out.println("こんにちは．学籍番号" + id + "の" + name + "です．"
				+ "よろしくお願いします．");
	}

	/** (2) 卒業報告をする */
	public void graduate() {
		System.out.print(name + "の単位数は，" + credit + "です．");
		if (credit >= REQUIREMENT_OF_GRADUATION) {
			System.out.println("卒業要件を超えているので，卒業できます．");
		} else {
			System.out.println("卒業要件に足りないので，卒業できません．");
		}
	}

	/** (3) 単位数を追加する */
	public void addCredit(int amount) {
		System.out.println(name + "の単位数を " + amount + "追加します．");
```

```
        credit = credit + amount;
    }

    /* ---------------- 以下，アクセサ (getter/setter)-----------*/

    /**
     * 卒業要件単位数を取得する（getterの命名法に伴って，メソッド名変更）
     * (REQUIREMENT_OF_GRADUATIONのgetter)
     */
    public static int getRequirementOfGraduation() {
        return REQUIREMENT_OF_GRADUATION;
    }


    /**
     * 学籍番号を取得する．
     * (フィールドidのgetter)
     */
    public String getId() {
        return id;
    }

    /**
     * 学籍番号をセットする．空文字列の場合には，"99999999X"をセットする．
     * (フィールドidのsetter)
     */
    public void setId(String id) {
        if (id.equals("") || id == null) {
            System.out.println("無効なIDです．99999999Xをセットします．");
            this.id = "99999999X";
        } else {
            this.id = id;
        }
    }

    /**
     * 名前を取得する(フィールドnameのgetter)
     */
    public String getName() {
        return name;
    }

    /**
     * 名前をセットする．空文字列の場合には，"名無し"をセットする．
     * (フィールドnameのsetter)
     */
    public void setName(String name) {
        if (name.equals("") || name == null) {
            System.out.println("無効な名前です．「名無し」をセットします．");
            this.name = "名無し";
        } else {
            this.name = name;
        }
    }

    /**
     * 単位数を取得する（フィールドcreditのgetter）
     */
    public int getCredit() {
        return credit;
    }

    /**
     * 単位数をセットする．負の場合は0に補正する．
     */
    public void setCredit(int credit) {
        if (credit < 0 ) {
            System.out.println("負の単位数は認められません．0にセットします．");
            this.credit = 0;
        } else {
            this.credit = credit;
        }
    }
```

```
        /**
         * 文字列表現.  "12345678X(タブ)中村(タブ)146単位"のような感じで表現する
         */
        @Override
        public String toString() {
                //String.format() はSystem.out.printf()とほぼ同じだが文字列として返すメソッド.
                return String.format("%s\t%s\t%4d単位", getId(), getName(), getCredit());
        }

}
```

Main.java (カプセル化後のStudent.javaを操作)

```
import java.util.ArrayList;

public class Main {

        public static void main(String[] args) {

                // 空の学生インスタンスを生成する.
                Student s = new Student();

                // フィールドに値を直接代入
                //（カプセル化により出来なくなっている）
                /*
                s.id = "";
                s.name = "";
                s.credit = -136;
                */

                //意地悪な値をsetしようとしてもOK.
                s.setId("");
                s.setName("");
                s.setCredit(-136);

                s.hello();
                System.out.println("-------------------------------------------------");


                // フィールドの値を直接読み出し
                //（カプセル化によりできなくなっている）
                /*
                System.out.printf("学番:%s, 名前：%s, 単位数：%s\n",
                                s.id, s.name, s.credit);
                */


                //toString()により，学生インスタンスをそのままprintln()可能.
                System.out.println(s);

                System.out.println("-------------------------------------------------");

                //(練習)ArrayListで複数の学生インスタンスを管理してみる
                ArrayList<Student> list = new ArrayList<Student>();
                //生成して同時にリストに追加
                list.add(new Student("12345678X", "中村", 136));
                list.add(new Student("18512245X", "前田", 86));
                list.add(new Student("18416623X", "中谷", 128));
                list.add(new Student("19952341X", "佐伯", 36));

                //現状を表示
                for (Student st: list) {
                        System.out.println(st);
                }
                System.out.println("-------------------------------------------------");

                //2単位足してみる
                for (Student st: list) {
                        st.addCredit(2);
```

```
                }
                System.out.println("-----------------------------------------------");

                //現状を表示
                for (Student st: list) {
                        System.out.println(st);
                }
        }
}
```

解説

- Studentクラスの3つのフィールド(id, name, credit)は，カプセル化の指針によりprivateに変更
- 3つのフィールドのためのsetter/getterを作成
- setterでは，無効な値が来た時の例外処理を書いている
- 引数付きコンストラクタではsetterを呼び出し，例外処理が効くように変更
- toString()は，インスタンスのフィールドの値を1行の文字列で返すようにしている．これにより，学生インスタンスをprintln()できるようになる（詳細は第6回説明）．

クラス図

第4回で述べたように，オブジェクト指向アプローチでは，複数のクラスに責務を分担し，これらが協調して全体の問題を解決するようにプログラミングを行う．しかしながら，クラスの数が多くなってくると，クラスのメンバやクラス間の関係を把握することが難しくなってくる．

クラス図の導入

この問題を解決するために，ここでは**クラス図**について学ぶ．クラス図は，各クラスのメンバとクラス間の関係を図で表すもので，プログラム全体のクラスの静的な構造を俯瞰できる．クラス図は，本来コーディング前のクラス設計の段階で作成するものであるが，現状のプログラムの分析，可視化の用途にも使える．クラス図はUMLというソフトウェア・モデリング言語の1つである．

基本

クラスは長方形で表現する．長方形の中は水平線で3つに分割され，各領域に以下を順に書く： クラス名，フィールド名，メソッド名．クラス図で必須なのはこの長方形と名前だけである．以下は必要に応じて，付け加える．

アクセス権

アクセス権に応じて，名前の前に以下の記号を書く： +(publicメンバ), - (privateメンバ), # (protectedメンバ)

型・戻り値

名前の後ろに：をつけて書く．

static

クラス(static)フィールド，クラス(static)メソッドは，名前に下線をつける（例： staticName）．

abstract (第6回で説明)

抽象(abstract)クラス，抽象(abstract)メソッドは，名前を斜体で書く（例：*abstractName*）

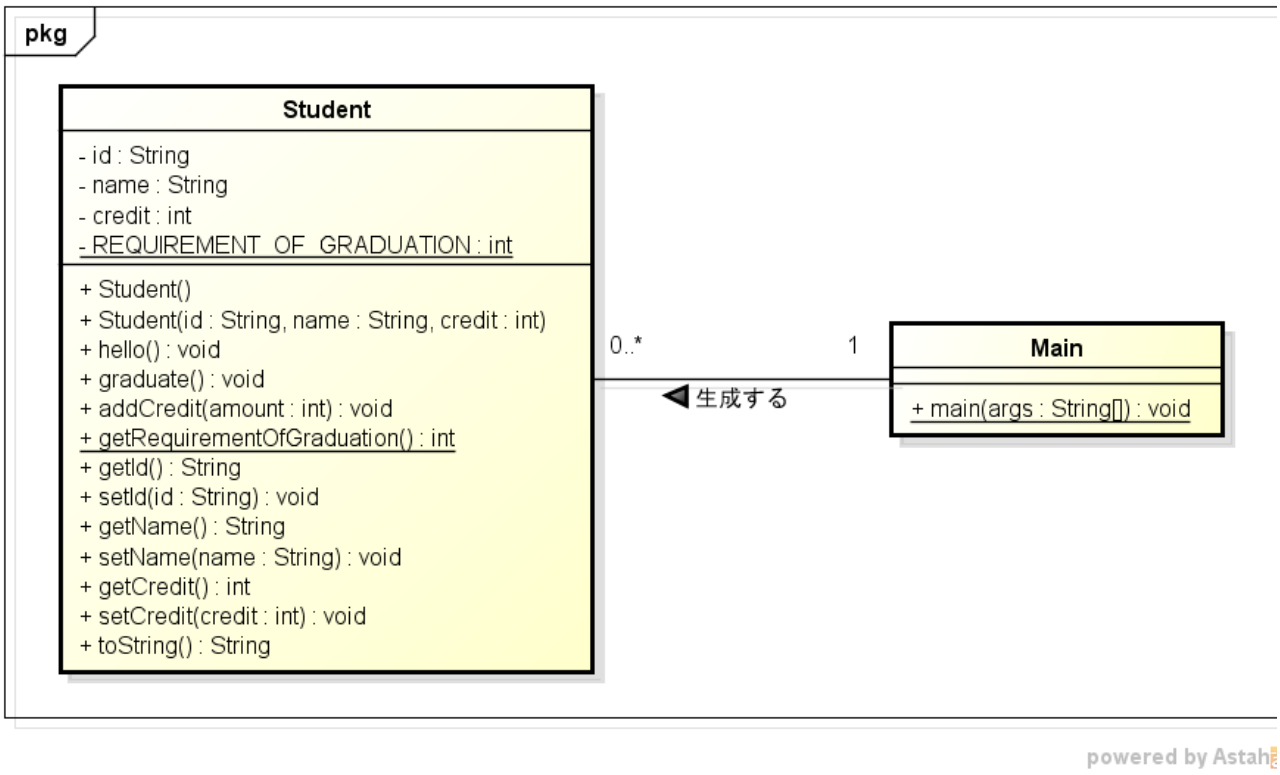
関連

クラス間に線を引いて関連を表す．関連の向きを表す▼をつけることができる．

例題1： はじめてのクラス図

上記で述べたカプセル化後のStudent.java と Main.javaをクラス図で記述すると以下のようになる．

クラス図の作成は，Astah* Communityというツールを使っている．演習室のPCからは，Finderのアプリケーションから見つけることができる．Windows版もあるので，興味のある人は自分のPCにもインストールしてみよう．



解説

- Studentと書かれている箱がStudentクラスを表す．3段組の上から，クラス名，フィールド(属性)，メソッド(操作)の欄を表している．
- メンバの名前の右に型が書かれている．コンストラクタはその定義により戻り型を持たない．
- メンバの前の-や+は可視性を表す．フィールドはprivate，メソッドはpublicになっていることが読み取れる．
- 下線が引いてあるのはstaticメンバを表す．
- 今回は練習のためgetter/setterもすべて書いてあるが，煩雑になるのでアクセサは省略される場合が多い（フィールドから機械的に生成できるため）．
- MainからStudentへの線は，**関連**を表す．この例では，Mainの中で，Studentインスタンスを複数生成してテストしていることから，このように書いている．
- 関連の端の数字は，**多重度**を表す．一方のクラスのインスタンスを1とするとき，他方のクラスのインスタンスがいくつ関連付けられるかを表している．
  - Mainクラスから複数のStudentインスタンスを作成しているので，Studentの関連端に0..* (0以上たくさん)を書いている．
  - Studentインスタンス1つから見たMainの数は1なので，Mainの関連端に1を書いている．
- 一番外枠pkgはパッケージを表す．今は気にしなくて良い．
- 各メンバの意味や中身の実装についてはクラス図には書かない．そのため，メンバには分かりやすい名前付けが重要となる．

関連

クラス間の関連には，大きく以下の3種類がある．

集約 (Aggregation)

あるクラスAが他のクラスBの構成要素になっている時 (B has A または A is a part of B)，AからBに◇のついた矢印を引く．このような「持っている (has-a)」という関係を集約 (Aggregation)という．また，インスタンスの多重度を表す数字を矢印の両端に書くことがある(*は任意の個数を表す)．

図1: 集約の例

継承 (Inheritance)

あるクラスBが他のクラスAの性質（種類）を引き継いだ拡張である時 (B is a A または B is a kind of A)，BからAに△のついた矢印を引く．このような「である (is-a)」という関係を継承 (Inheritance)という．継承の場合は多重度を書かない．継承に関する説明は第6回で行う．

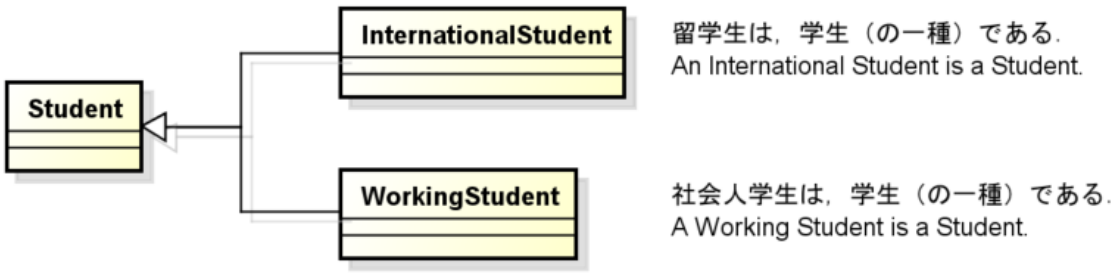


図２：継承の例

一般関連

集約，継承以外の任意の関連．クラス間に線を引いて，関連の名前を付ける．向きを表す▲をつけることができる．多重度は必要があれば書く．

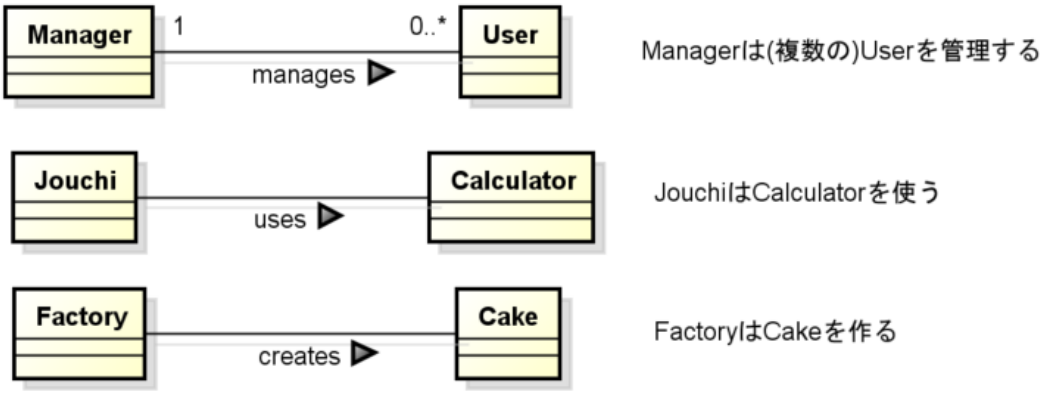


図３：一般関連の例

例題２：友達リスト拡張版

第3回演習・問２の友達リストの問題を，もう少し現実的に拡張してみよう．

- 以前は，管理する友達の情報は名前のみだったが，今回は名前(name)，電話番号(phone)，メールアドレス

(email)に拡張する．

- 友達の追加は， 名前，電話番号，メールアドレスを入力して登録する．
- 友達の削除は，番号を指定して削除する．
- 登録されている友達のリスト表示は，以前のまま番号と名前のみの表示で良いが，今回は番号を指定して友達の詳細情報（名前，電話番号，メールアドレス）を表示できるようにしたい．

以上の要件に基づいて，クラス設計を行った例を以下に示す．問題に登場するクラスと，各クラスへの責務の分担を読み解いてみよう．（自明なコンストラクタやsetter/getter等は省略している）

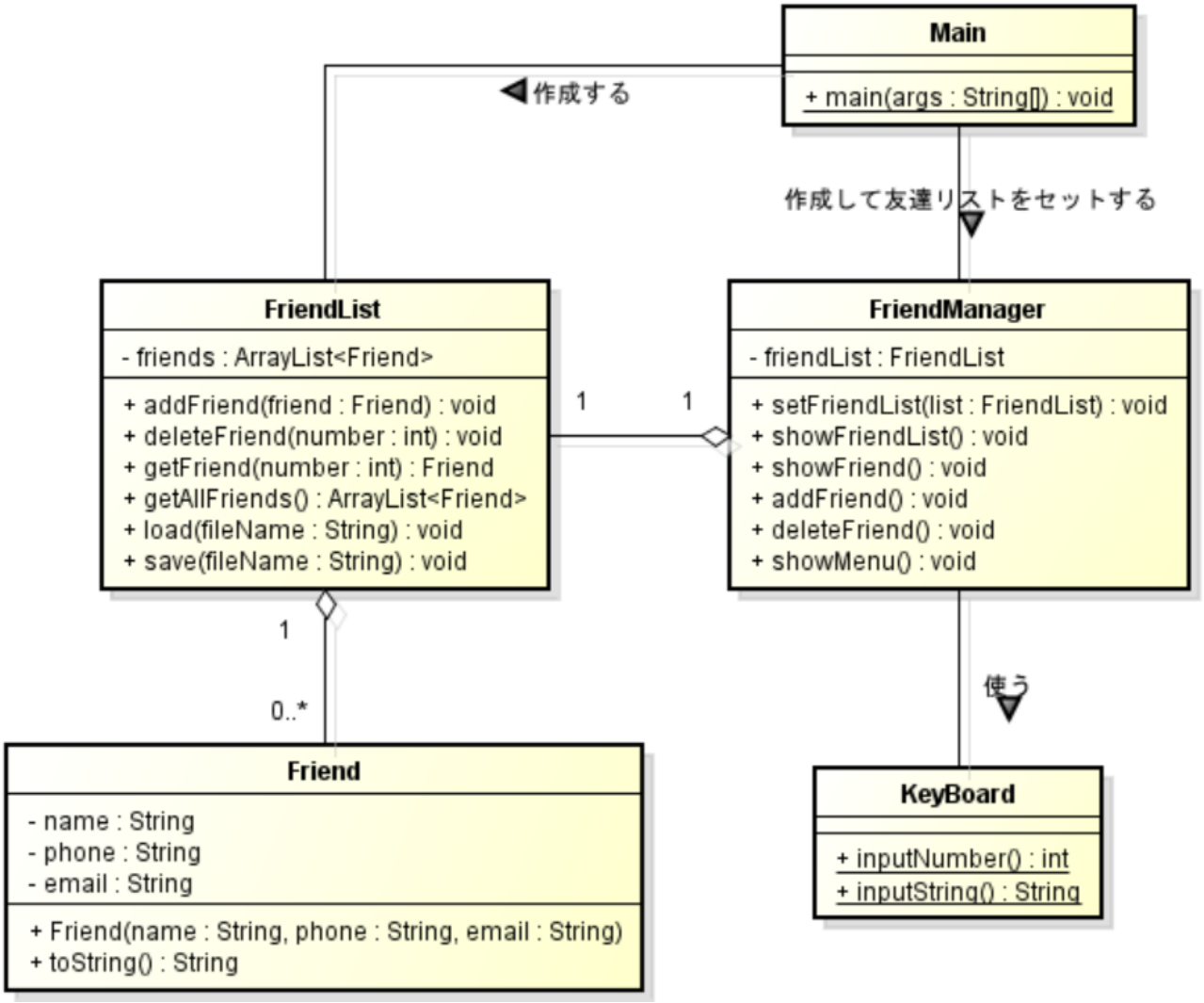


図４：友達管理アプリのクラス図

演習問題

- 第5回演習問題へ

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

Last updated: 12/18/2018 08:52:33