JAVA入門

今回の内容

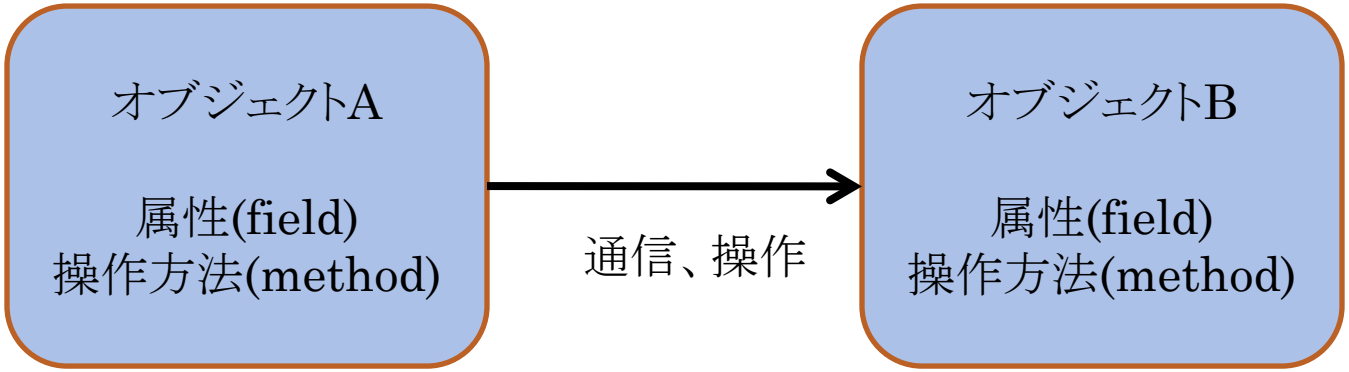
- グラフとオブジェクト指向プログラミング
  - Javaを使う理由
- Javaの基本
  - Javaのライブラリ
  - 開発・実行
- クラスの再利用
  - クラス継承
  - 抽象クラス
- 開発の要点

グラフを記述するには

- 頂点(Vertex)と弧(Arc)、その間の関係
- 素直にデータ構造として表現したい
  - グラフは、頂点と弧の集合
  - 弧から始点と終点を得る
  - 頂点から、その頂点を始点とする弧の集合を得る
- 頂点と弧をモノ (object) として捉える
  - モノを中心にプログラムを考える枠組みが欲しい
  - オブジェクト指向プログラミング

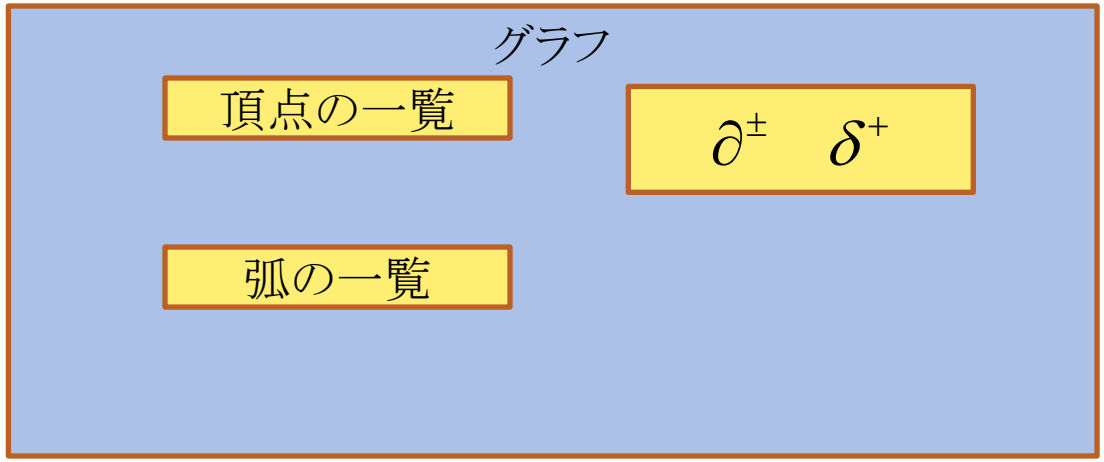
オブジェクト指向(Object Oriented)

- もの(object)の操作・動作を中心に考える
  - 操作や動作を日本語で考え、出てくる名詞に注目する
- オブジェクトの構成
  - 属性(field):データなど
  - 操作(method)



グラフをオブジェクト指向プログラミングで考える

- グラフの構造を表すデータ構造
  - グラフ、頂点、弧
- 階層的データ
  - グラフの要素としての頂点と弧
  - 頂点に接続している弧のリスト
  - 弧の両端の頂点



- 各データごとの操作
  - 弧に値を設定する
  - 探索問題：頂点や弧に印を付ける
- データのカプセル化
  - グラフとしての整合性を維持
  - 頂点や弧に属性等を追加
- 型の継承と拡張
  - 弧に「流れ」の属性を付けて拡張
- グラフの可視化

様々なOOP言語

- Smalltalk80
  - Xerox, Palo Alto研究所
- C++
  - B. Stroustrup
  - C にOOPを導入
- Java
  - Sun Microsystems (Oracle)
- Ruby
  - まつもとひろゆき
  - スクリプト言語

C++ではなくJAVAを使う理由

- 豊富なユーティリティー
  - java.util.ArrayListなど
- 使い易い開発環境(IDE)
  - NetBeans、Eclipse
- 多数のOSで使える
  - Windows、Linux、Solaris
  - OSに依存しない実行形式
- GUI開発が容易
  - IDEを使うと簡単

JAVAの基本

- 全てがクラス
- 開始点となるクラスが必要
  - public static void main(String[] args)メソッドから始まる
  - mainは主となるクラスを起動するだけ
- コンストラクタメソッド
  - クラスと同じ名前のメソッド
- デストラクタは無い
  - 自動ガベージコレクション

- 一つのクラスで一つのファイルが基本
  - ファイル名はクラス名と同じ
- ヘッダファイルが無い
  - ライブラリはimport文を使う
- C/C++のポインタは無い
  - 原始型は値代入
  - クラスオブジェクトは参照

- 文法はだいたいC++と同じ
- 原始型はint、double、char、booleanなど
- 原始型に対応したクラスがある
  - Integer、Double、Character、Booleanなど
- 文字列Stringや原始型の配列はクラス
- ポインタが無い
- デストラクタは書かない
  - 不要なオブジェクトは自動で削除される
- クラスは階層化され、パッケージになっている

```
package StudentSample;
public class Student {
    //クラス内のフィールド
    private String name=null;//名前
    private int studentID=0; //学生番号
    private int record=0;    //点数
    /**
     * コンストラクタ : インスタンスを生成する
     */
    public Student(String name, int studentID) {
        this.name=name;
        this.studentID=studentID;
    }

    /**        取得メソッドと設定メソッド        **/
    public int getStudentID() {return studentID;}

    public String getName() {return name;}

    public int getRecord() {return record;}

    public void setRecord(int record) {
        this.record = record;
    }
}
```

クラス宣言

クラス内フィールド
クラス内のデータ

コンストラクタ
クラスインスタンス生成

メソッド
クラスインスタンス操作

便利なライブラリ

- オンラインマニュアル
  - http://docs.oracle.com/javase/7/docs/api/
- 基本的なクラス:java.lang
- 入出力:java.io
- コレクション(リストなど):java.util
- 基本GUI:java.awt
- 拡張GUIセットSwing:javax.swing

開発環境

- http://www.oracle.com/technetwork/java/javase/downloads/index.html
- NetBeans
  - http://www.netbeans.org/index_ja.html
- プロジェクト管理
- UMLとの連携
- メソッド名の補完
- パーツを使ったGUI構築
- CVS等を使ったバージョン管理

プログラム開発の手順

- 作業ディレクトリを決める
  - デフォルトでは~/Documents/NetBeansProjects
- NetBeansを起動
- 「ファイル」→「新規プロジェクト」
- プロジェクトウィンドウ内で
- プロジェクト名→「ソースパッケージ」→「デフォルトパッケージ」で右ボタン「新規」

新しいクラスを作る

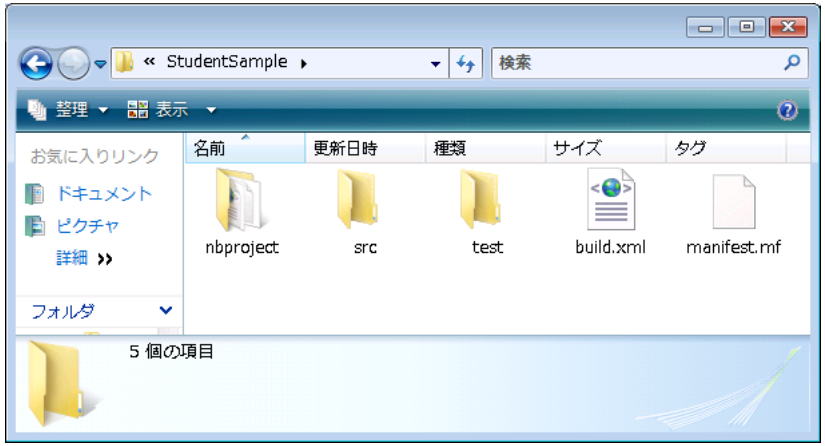
- GUIの無い主クラス
  - 「Java 主クラス」
- GUIのある主クラス
  - 「JFrameフォーム」
- テンプレートを上手に使う

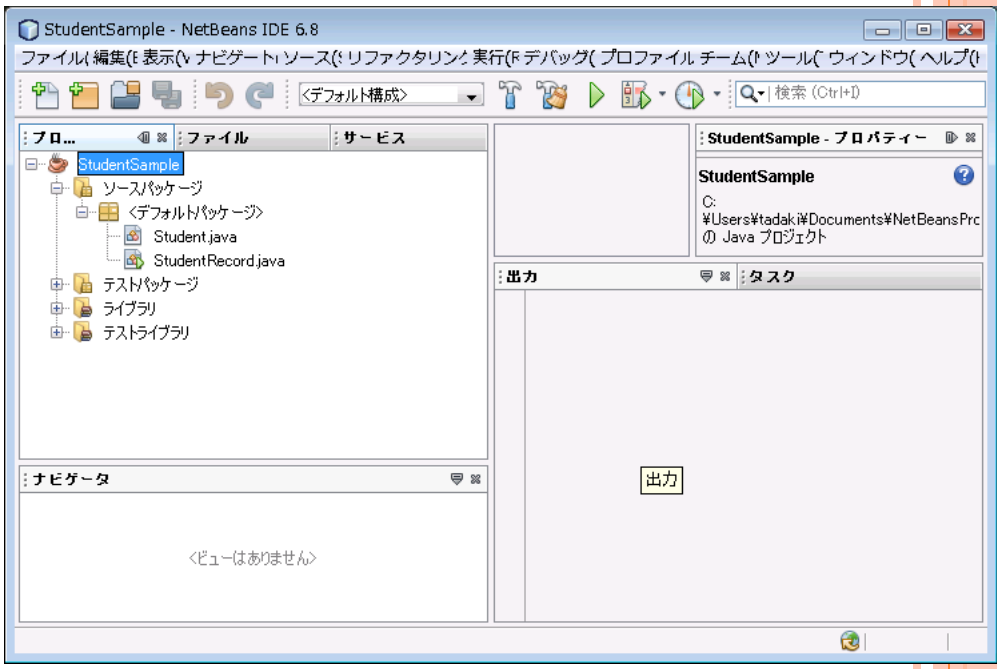
構築と実行

- プロジェクトウィンドウ内で
  - プロジェクト名→「プロジェクトを構築」
- プロジェクトウィンドウ内で
  - プロジェクト名→「プロジェクトを実行」
  - 主クラス名→「ファイルを実行」
- デフォルトでは、ファイルを保存すると、コンパイルする

サンプルプログラムを作成する場合
例：STUDENTSAMPLE

- プロジェクト「StudentSample」を作成する
- プロジェクトディレクトリの構造
  - 「src」の下に*.javaファイルをダウンロード
- プロジェクト内にクラスが表示される





OOPと開発効率

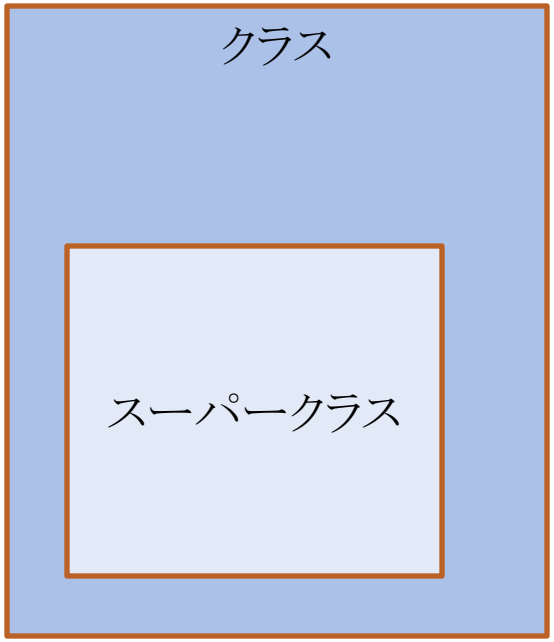
- OOPはプログラム開発効率を改善する
- カプセル化
  - クラス内部の構造を隠す
  - 変更をクラス内に止め、他に影響を与えない
- クラスの継承・再利用
  - 機能や属性を既存のクラスに追加する
- 抽象クラス
  - 機能や属性の似たクラスをグループ化する

クラスの継承・再利用

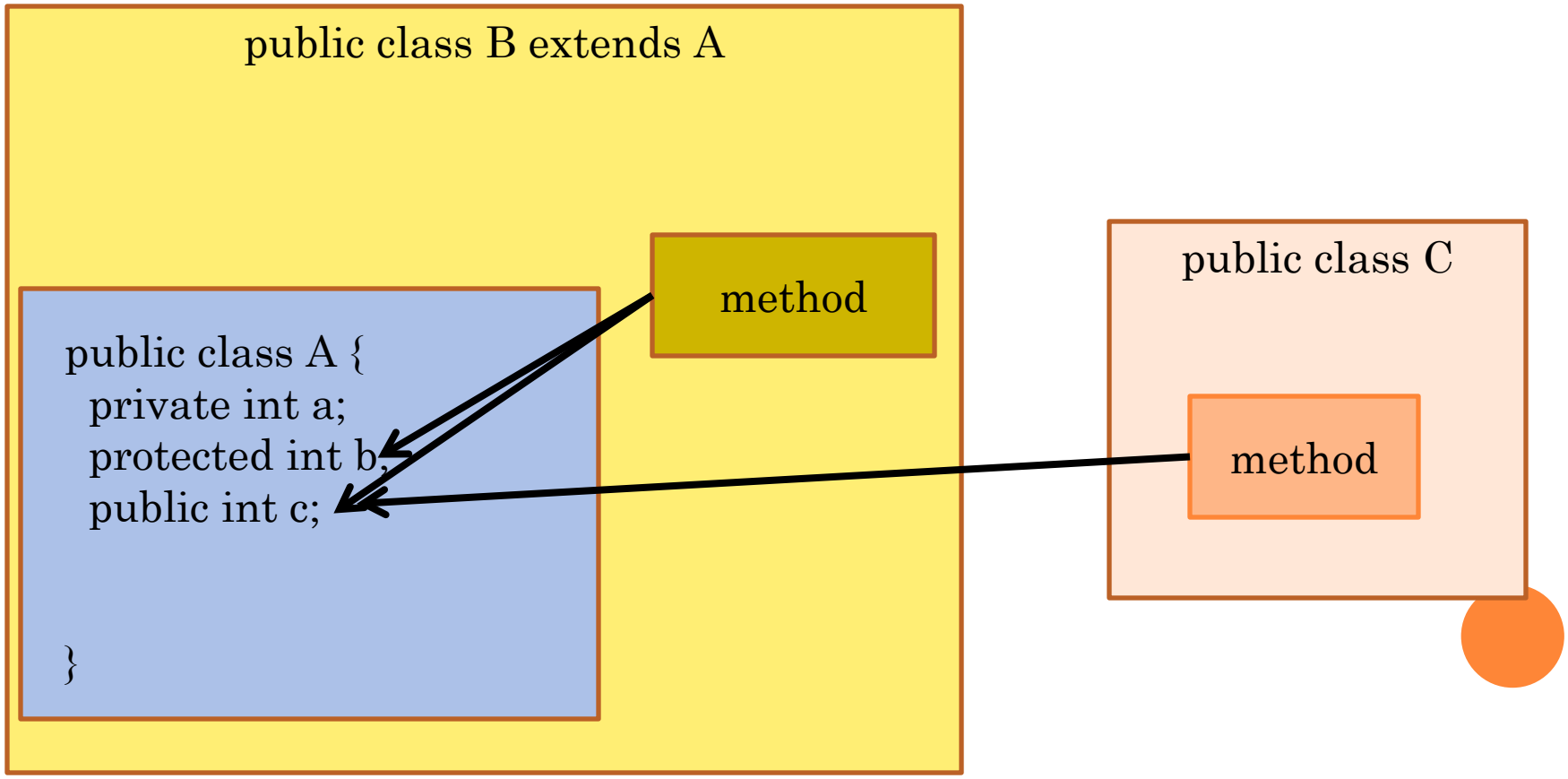
- 既存のクラスを継承して拡張
  - クラスの継承とインターフェイスの利用
- 既存のクラスとの調整をするクラスを作る
  - インターフェイス的な調整
- 既存のクラスを要素として持つクラスを作る

クラスの継承

- 出来上がったクラスの資産を生かす
- 標準的クラスの資産を生かす
- クラスの組に共通なデータや動作を定義する
- 一つのクラスしか継承できないことに注意



アクセス制限


public class B extends A
public class A {
private int a;
protected int b;
public int c;
}
method
public class C
method

Abstract classes

- 基本となるデータ構造とメソッドを定義
- メソッドの一部は実装が定義されていない
  - abstract method
- 継承クラスを定義して使う
- 例
  - java.util.AbstractList
  - 上記の実装の一つがjava.util.ArrayList

Interfaces

- abstract methodのみで構成されている
- アクセス方法だけが指定されている
  - 他のプログラムからの見え方を規定

```
class A implements インターフェイス{
}
```

- 例
  - java.lang.Runnable
  - メソッドrun()が定義されている
  - スレッドからの呼び出しに使う

例：COMPARABLEインターフェイス

- 大小関係があるインスタンスを表す
  - 順序付けることができる
- 必ずメソッド int compareTo()を実装しなければならない。
  - 引数と比較して、0または±1を返す
- ソートプログラムは、Comparable インターフェイスを持ったクラスをcompareTo()を使って並べ替える。
  - クラスの中の構造は知らなくて良い

プログラム開発の要点

- 開発・保守コストを下げる
- クラスの再利用
  - ルーチン化したコードを再利用
  - 他の人のノウハウを借用
- 分かりやすい構成
  - 自分にも他人にもわかるように
  - 修正箇所の限定
  - 修正の影響範囲を明確化

プログラム開発の要点2

- アルゴリズムをデータの詳細と切り離す
  - ソートのアルゴリズムは、ソートされるデータの詳細とは関係ない
  - スレッドプログラムは、各スレッド内で何をしているかと関係ない
- 問題をオブジェクトの運動として捉える
  - 小さなオブジェクトへ分割
  - 小さなオブジェクトならば、その役割が明確になる

Student.java

```java
package StudentSample;
/**
 * Student.java
 * Created on 2007/04/15, 11:34
 * 生徒のクラス
 * @author tadaki
 */
public class Student {
    //クラス内のフィールド
    private String name=null;//名前
    private int studentID=0; //学生番号
    private int record=0;    //点数
    /**
     * コンストラクタ : インスタンスを生成する
     */
    public Student(String name, int studentID) {
        this.name=name;
        this.studentID=studentID;
    }

    /**        取得メソッドと設定メソッド      **/
    public int getStudentID() {return studentID;}

    public String getName() {return name;}

    public int getRecord() {return record;}

    public void setRecord(int record) {
        this.record = record;
    }
}
```

```java
package StudentSample;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/*
 * StudentRecord.java
 *
 * Created on 2007/04/15, 11:34
 * @author tadaki
 */
public class StudentRecord {

    private List<Student> students = null;//生徒一覧
    private String names[] = {
        "Aoyama", "Asou", "Baba", "Chou", "Egashira",
        "Eto", "Funaki", "Goto", "Gunji", "Hara", "Hashimoto",
        "Ikeuchi", "Ito", "Jo", "Kayama", "Mori", "Naito", "Tada",
        "Yamada", "Yoshida"

    };

    /** コンストラクタ */
    public StudentRecord() {
        //生徒一覧を初期化
        students = Collections.synchronizedList(new
ArrayList<Student>());
        //登録
        for (int i = 0; i < names.length; i++) {
            Student s = new Student(names[i], 1000 + i);
            s.setRecord((int) (100 * Math.random()));
            students.add(s);
        }
    }

    public void listStudents() {
        int max = 0;
        Student best = students.get(0);

        //拡張されたforループ
        for (Student s : students) {
            int r = s.getRecord();
            System.out.print(String.valueOf(s.getStudentID())
```

```
                    + ":" + s.getName() + ":");
            System.out.println(String.valueOf(r));
            if (r > max) {
                max = r;
                best = s;
            }
        }
        System.out.println();
        System.out.print("Best is ");
        System.out.print(String.valueOf(best.getStudentID())
                + ":" + best.getName() + ":");
        System.out.println(String.valueOf(best.getRecord()));
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        StudentRecord studentRecord = new StudentRecord();
        studentRecord.listStudents();
    }
}
```

Student.java

```java
package StudentSample2;

/**
 * 生徒のクラス
 * Comparableインターフェイスの例
 * @author tadaki
 */
public class Student implements Comparable<Student> {

    private String name = null;//名前
    private int studentID = 0; //学生番号
    private int record = 0;    //点数

    /**
     * コンストラクタ
     * @param name 名前
     * @param studentID 学生番号
     */
    public Student(String name, int studentID) {
        this.name = name;
        this.studentID = studentID;
    }

    /**
     * 学生番号取得
     * @return  取得した学生番号
     */
    public int getStudentID() {
        return studentID;
    }

    /**
     * 名前取得
     * @return 取得した名前
     */
    public String getName() {
        return name;
    }

    /**
     * 得点取得
     * @return 取得した得点
     */
    public int getRecord() {
```

```
        return record;
    }

    /**
     * 得点設定
     * @param record 設定する得点
     */
    public void setRecord(int record) {
        this.record = record;
    }

    @Override
    /**
     * Student インスタンスの比較
     * インターフェイスComparableで必須
     */
    public int compareTo(Student o) {
        int k = 1;
        if (this.getRecord() < o.getRecord()) {
            k = -1;
        }
        return k;
    }
}
```

StudentRecord.java

```java
package StudentSample2;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 *
 * @author tadaki
 */
public class StudentRecord {

    private List<Student> students = null;//生徒一覧
    /** 名前一覧 */
    private String names[] = {
        "Aoyama", "Asou", "Baba", "Chou", "Egashira",
        "Eto", "Funaki", "Goto", "Gunji", "Hara", "Hashimoto",
        "Ikeuchi", "Ito", "Jo", "Kayama", "Mori", "Naito", "Tada",
        "Yamada", "Yoshida"
    };

    /** コンストラクタ */
    public StudentRecord() {
        //生徒一覧を初期化
        students = Collections.synchronizedList(new
ArrayList<Student>());
        //登録
        for (int i = 0; i < names.length; i++) {
            Student s = new Student(names[i], 1000 + i);
            s.setRecord((int) (100 * Math.random()));
            students.add(s);
        }
    }

    /**
     * 学生一覧印刷
     */
    public void listStudents() {
        //拡張されたforループ
        for (Student s : getStudents()) {
            System.out.print(String.valueOf(s.getStudentID())
                    + ":" + s.getName() + ":");
            System.out.println(String.valueOf(s.getRecord()));
        }
```

```
    }

    /**
     * 学生一覧取得
     * @return 学生一覧のVector
     */
    public List<Student> getStudents() {
        return students;
    }

    /**
     * ソートの実行
     * @param <T> Comparableインターフェイスを実装したクラス
     * @param t Vector<T>
     */
    public static <T extends Comparable<T>> void sort(List<T> t) {
        for (int i = t.size(); i > 0; i--) {
            for (int j = 0; j < i - 1; j++) {
                if (t.get(j).compareTo(t.get(j + 1)) > 0) {
                    T c = t.get(j);
                    t.set(j, t.get(j + 1));
                    t.set(j + 1, c);
                }
            }
        }
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        StudentRecord studentRecord = new StudentRecord();
        StudentRecord.sort(studentRecord.getStudents());
        studentRecord.listStudents();
    }
}
```