情報処理概論
(Basic Theory of Information Processing)

第１２回：クラスとインスタンス

概要

- クラスの宣言
- フィールドの宣言，メソッドの定義
- オブジェクトの作成，コンストラクタ
- オーバーロード
- クラスの継承 `extends`
  - メソッドのオーバーライド
  - ポリモフィズム
- 演習

14　クラスの宣言 (p.270)

クラス：オブジェクトの設計図

オブジェクト	＝	属性	＋　操作
		データ	＋　処理
		フィールド	＋　メソッド

クラス宣言の中ではフィールドの宣言とメソッドの定義を行う.

```
class クラス名 {
   フィールドの宣言
   ・・・・・・・・
   ・・・・・・・・

   メソッドの定義
   ・・・・・・・・
   ・・・・・・・・

}
```

- クラス名は識別子で，識別子の命名規則に従う.

14.1　フィールドの宣言 (p.271)

- 変数宣言と変わらない.
- フィールド名は識別子で，識別子の命名規則に従う.

フィールドの宣言の例

```
class StudentBasic {
  int stdNo;
  int pMath, pPhys;
  String name;
  double height;
}
```

14.1.1　クラス型変数 (オブジェクト参照型変数)

- クラス型変数：
  指定したクラスにしたがって生成されたオブジェクトを参照できる。
- 上の例では，クラスString型の変数nameを宣言している。

14.1.2　フィールドの可視性 (p.278)

修飾子	アクセス可能なクラス
なし	同じパッケージに属するクラスのみ
`private`	そのクラスのみ
`protected`	同じパッケージ内のクラス，またはサブクラスのみ
`public`	基本的に全てのクラスから

- **クラスが条件を満たしていれば，オブジェクトは異なっていても良い．**

```
class StudentAccess {
  public int stdNo
  private int pMath, pPhys;
  protected String name;
  double height;
}
```

14.2　メソッドの定義 (復習)

メソッド定義の例

```
class StudentMethod {
  int stdNo, pMath, pPhys;
  String name;
  double height;

  ⇓ メソッド setPoint() の定義の開始
  void setPoint(int inPMath, int inPPhys) {
    pMath = inPMath;
    pPhys = inPPhys;
  } ⇐ここまで

  ⇓ メソッド avePoint() の定義の開始
  double avePoint() {
    return (pMath + pPhys) * 0.5;
  } ⇐ここまで
}
```

- メソッド setPoint()
  - 引数：int 型の inPMath と inPPhys
  - 戻り値：なし (void)
  - 機能：フィールド pMath と pPhys に，それぞれ，引数 inPMath と inPPhys の値を代入する。
- メソッド avePoint()
  - 引数：なし
  - 戻り値：double 型
  - 機能：フィールド pMath と pPhys の平均を計算して，戻り値として返す。

14.3 メソッドの可視性

修飾子	アクセス可能なクラス
なし	同じパッケージに属するクラスのみ
private	そのクラスのみ
protected	同じパッケージ内のクラスまたは，サブクラスのみ
public	基本的に全てのクラスから

プログラム例

```
class StudentAccessMethod {
  public int stdNo;
  private int pMath, pPhys;
  protected String name;
  double height;
  private boolean open = false;

  protected double avePoint() {
    if (open) return (pMath + pPhys) * 0.5;
    else return 0.0;
  }
  private void setPoint(int inPMath, int inPPhys) {
    pMath = inPMath; pPhys = inPPhys;
  }
  public double getHeight() {
    if (open) return height;
    else return 0.0;
  }
}
```

14.4　オブジェクト (参照型の) 変数の宣言

- 作成したオブジェクトは，**クラス型 (オブジェクト参照型) 変数**を使って参照することができる.
- その変数には，オブジェクトの参照情報が格納されている。
  参照情報：オブジェクトの格納場所，メモリーのアドレスのようなもの
  例：クラス StudentBasic のオブジェクトのための参照型変数 x を宣言する.

```
StudentBasic x;
```

- クラス StudentBasic はあらかじめ宣言されていなくはいけない.

14.5　オブジェクトを作成する (p.270)

- 宣言されたクラスのインスタンスであるオブジェクトを作成する。
- new キーワードを使う.
  例：クラス StudentBasic のオブジェクトをコンストラクトし，オブジェクトの参照型変数 x に代入する.

```
x = new StudentBasic();
```

- StudentBasic() は，**コンストラクタ**と呼ばれるメソッドである (後述).

14.6　フィールドへのアクセス (p.274)

- オブジェクトのフィールドの値を，外から参照・変更する．
  オブジェクト名.フィールド名
  として，参照・変更ができる．
- あるオブジェクトのメソッドが，そのオブジェクトのフィールドを参照する場合は，
  オブジェクト名. は省略できる．

例：

```
class StudentBasic {
  int stdNo;
  int pMath, pPhys;
  String name;
  double height;
}
```

のフィールドに対してアクセスする。

```
public class MainStudent {
  public static void main(String args[]) {
    StudentBasic stda = new StudentBasic();
    StudentBasic stdb = new StudentBasic();

    stda.stdNo  = 4;
    stda.name   = "花澤香菜";
    stda.pMath  = 90;
    stda.pPhys  = 92;
    stda.height = 156.7;

    stdb.stdNo  = 2;
    stdb.name   = "野上ゆかな";

    System.out.println("学籍番号：" + stda.stdNo + ", 名前：" + stda.name
        + ", 数学：" + stda.pMath + ", 物理：" + stda.pPhys
        + ", 身長：" + stda.height);
  }
}
```

14.7　メソッドの利用 (p.283)

- オブジェクトのメソッドを，別のクラスのオブジェトで呼び出す.
  オブジェクト名. メソッド名 (実引数をカンマ区切りで列挙)
  として，呼び出すことができる (実引数はなくてもよい).
- 実引数を使って値を渡すことができる.
  - **実**引数：メソッド呼び出し側の引数.
  - **仮**引数：メソッド定義側の引数.

```
public class MainStudentMethod {
  public static void main(String args[]) {
    StudentMethod stda = new StudentMethod();
    StudentMethod stdb = new StudentMethod();
    stda.stdNo  = 4; stda.name   = "花澤香菜  ";
    stdb.stdNo  = 2; stdb.name   = "野上ゆかな";
    stda.setPoint(90, 92);
    stdb.setPoint(80, 87);
    System.out.println("名前:" + stda.name + " 平均点:" + stda.avePoint());
    System.out.println("名前:" + stdb.name + " 平均点:" + stdb.avePoint());
  }
}
```

14.8 コンストラクタ (p.280)

- オブジェクトを作成するための特別な**メソッド**である。
  (領域の確保，初期値の設定等を行う。)
- 名前はクラスと同じ。
- 戻り値はない**(void の指定も不用)**.
- 特に定義しなくても，クラスを宣言すれば暗黙的にコンストラクタが定義される.
  - そのオブジェクトのフィールドのメモリー領域を確保する.
- 明示的に定義することもできる.
  そのコンストラクタには次のような処理を記述する.
  - フィールドなどに初期値を設定する.
  - 部品とするオブジェクトを作成する.
  - 配列の領域を確保する.
- キーワード`new`を使って，コンストラクトする.

  `new` クラス名(実引数，実引数，...)

  引数はないときもある.

  `new` クラス名()

(参考：C++の場合はデコンストラクタが必要になるが，javaでは不用である．)

例

```
class StudentConstructor {
  int stdNo, pMath, pPhys;
  String name;
  double height;

   ⇓ コンストラクタの定義のはじまり
  StudentConstructor(int inStdNo, String inName, double inHeight) {
    stdNo  = inStdNo;
    name   = inName;
    height = inHeight;
  }    ⇐ コンストラクタの定義の終わり
  void setPoint(int inPMath, int inPPhys) {
    pMath = inPMath;
    pPhys = inPPhys;
  }
  double avePoint() {
    return (pMath + pPhys) * 0.5;
  }
}
```

コンストラクタの利用例

```
public class MainStudentConstructor
  public static void main(String args[])
    StudentConstructor stda
                = new StudentConstructor(4, "花澤香菜  ", 156.7);
    StudentConstructor stdb
                = new StudentConstructor(2, "野上ゆかな", 160.0);

    stda.setPoint(90, 92);
    stdb.setPoint(80, 87);

    System.out.println("名前:" + stda.name + "  平均点:" + stda.avePoint());
    System.out.println("名前:" + stdb.name + "  平均点:" + stdb.avePoint());
```

14.9 デフォルトのコンストラクタ (p.281)

- コンストラクタを明示的に記述しなくても，特に何もしないコンストラクタが定義される。

```
class StudentBasic {
  int stdNo;
  int pMath, pPhys;
  String name;
  double height;

  public StudentBasic() {
  }
}
```

- 明示的なコンストラクタが1つでも定義されると，デフォルトのコンストラクタは定義されなくなる。
  `StudentConstructor stda = new StudentConstructor();`
  は，コンストラクタが存在しないというエラーになる。

14.10　コンストラクタなしでフィールドを初期化 (p.275)

- 変数宣言時に初期化できる。
  オブジェクトをコンストラクトするときに，値が代入される。

```
class StudentInit {
  int stdNo;
  int pMath, pPhys;
  String name;
  double height = 155.0;
}

public class MainStudentInit {
  public static void main(String args[]) {
    StudentInit std = new StudentInit();
    System.out.println("height = " + std.height);
  }
}
```

14.11　フィールドとローカル変数

- フィールド
  - **インスタンス変数**：
    - 自身のオブジェクトの中で参照することができる
      (そのオブジェクトの全てのメソッドから参照可能).
    - 領域は各オブジェクトごとに確保される.
    - アクセス制限の指定によっては，オブジェクト外部から参照することも可能
  - **クラス変数**：
    - そのクラスのすべてのオブジェクトから参照することができる.
    - 領域は，クラスで１つ分だけ確保される.
      １つのオブジェクトのメソッドが値を変えると，同じクラスの他のオブジェクトから見ても値が変わっている。
    - 修飾子`static`を付ける.
    - アクセス制限の指定によっては，オブジェクト外部から参照することも可能
    - クラス名.フィールド名で参照する.
- **ローカル変数**：
  - 定義されたメソッドの中だけで参照することができる.
  - **ローカル変数が優先：**フィールドと同じ名前のローカル変数がある場合は，ローカル変数が参照される。

– 仮引数もローカル変数として扱われる。

例：

```
class ClassVar {
  public static void main(String args[]) {
    ClassVarSub a = new ClassVarSub();
    ClassVarSub b = new ClassVarSub();
    a.print();   b.print();
    a.x = 3;
    ClassVarSub.y = 4;
    a.print();   b.print();
  }
}
class ClassVarSub {
  int x = 1;          ⇐ インスタンス変数
  static int y = 2;   ⇐ クラス変数
  void print() {
    System.out.println("x = " + x + " y = " + y);
  }
}
```

結果：

```
x = 1 y = 2
x = 1 y = 2
x = 3 y = 4
x = 1 y = 4
```

例：フィールドとローカル変数で同じ識別子の変数がある場合

```
class MainLocalvsField {
  public static void main(String args[]) {
    LocalvsField obj = new LocalvsField();
    obj.checkPrint();
    obj.checkPrint2();
  }
}
class LocalvsField {
  int x, y;
  LocalvsField() { x = 1; y = 2; }
  void checkPrint() {
    int x = 10;
    System.out.println("x = " + x + "  y = " + y);
  }
  void checkPrint2() {
    double y = 20.0;
    System.out.println("x = " + x + "  y = " + y);
  }
}
```

14.12 ローカル変数のスコープ (変数が有効な範囲)

- **スコープ**：プログラム内で，ある変数にアクセスできる範囲.
- フィールドは，オブジェクトあるいはクラス単位で決まる (既述).
- ローカル変数は，**宣言されてから，そのブロックの終わりまで**がスコープ

```
class Scope {
  public static void main(String args[]) {
    int x, y;       ⇐ x, yのスコープのはじまり
    x = 0; y = 4;
    {
      x = x + 1;
      int z = 4;    ⇐ zのスコープのはじまり
    //int y = 3;    ⇐ これはyの２重定義になってだめ
      x = y * x;
      System.out.println("x = " + x + " y = " + y + " z = " + z);
    }               ⇐ zのスコープの終わり
    System.out.println("x = " + x + " y = " + y);
  }                 ⇐ x, yのスコープの終わり
}
```

- 一般的には，ローカル変数はメソッドのはじめにまとめて定義することが一般的.
  ⇒ あちこちで定義すると，どこで定義したか分からなくなってしまうため.
- ループのための変数の宣言：

```
public class ScopeFor {
  public static void main(String args[]) {
    int sum = 0;
    for (int i = 1 ; i <= 10 ; ++i) {
      sum += i;
    }
    System.out.println("sum = " + sum);
  }
}
```

- 上の例では，for 文の中が変数 i のスコープになる。

- 値の交換のためだけに使う変数

```
public class ScopeSwap {
  public static void main(String args[]) {
    int a = 3, b = -1;
    double x = 2.0, y = -5.0;
    // aとbの値を交換する。
    {int t = a; a = b; b = t;}
    System.out.println("a = " + a + " b = " + b);
    // xとyの値を交換する。
    {double t = x; x = y; y = t;}
    System.out.println("x = " + x + " y = " + y);
  }
}
```

- 変数tは，交換のためのブロックがスコープになる。
- ブロックを出ると，また変数名tを他の用途で使うことができる。
  型が違っても良い。

14.13　定数 (値を変更しない変数)

- プログラムの実行中に値が変わらないもの.
- 変数に修飾子`final`を付加する.
- 値は初期化によってだけ設定できる.
- 同じクラスのオプジェクで共通となるので，一般にはクラス変数とする.
- `final`と`static`の順番はどちらでも良い.
- オブジェクト変数としても良いが，その場合はオブジェクトを作成してから参照する.

例：

```
class MainFinal {
  public static void main(String args[]) {
    Final t = new Final();
    System.out.println("t.Width = " + t.Width);
    //t.Width = 2;  // 値はセットできない。
  }
}
class Final {
  final static int Width  = 720;
  static final int Height = 480;
  final int Depth = 255;
  int setPoint(int x, int y, int val) {
    if (x < 0 || x > Width) return -1;
    if (y < 0 || y > Height) return -2;
    if (val < 0 || val > Depth) return -3;
    // set point routine
    return 0;
  }
}
```

14.14 メソッドのオーバーロード

- メソッドはその識別子ばかりでなく，引数の型を含めて区別される．

```
int abc(int i) {
    ·····
}

int abc(int i, int j) {
    ·····
}

int abc(double i) {
    ·····
}
```

と定義されていたとき，

- abc(1) では，1番目のメソッドが呼び出される．
- abc(1, 2) では，2番目のメソッドが呼び出される．
- abc(1.0) では，3番目のメソッドが呼び出される．

- 戻り値だけでは区別できない．
- 次の例はエラーになる．

```
int abc(int i) {
    ·····
}
double abc(int i) {
    ·····
}
```

- 次の例はＯＫである．

```
int abc(double i) {
    ·····
}
double abc(int i) {
    ·····
}
```

- **メソッドの識別子と引数の型並び**を，そのメソッドの**シグネチャ**と呼ぶ。

14.15　this (p.288)

- thisは自分のオブジェクトを示す。
  (普通に自分のフィールドやメソッドを参照する場合は省略できる。)
- 利用法：
  - コンストラクタのオーバーロードをする。
  - ローカル変数，パラメータとフィールドを区別する。
- thisキーワードを使って，自身のコンストラクターを呼び出すときは，最初の行におかなくてはいけない.
- 「オーバーロード」に類似した単語に**「オーバーライド」**があるので，注意すること。

例

```
class MainThis {
  public static void main(String args[]) {
    This t = new This(2, 1.5);
    t.printField();
  }
}
```

```
class This {
  int    n;
  double stx, sty;
  This(int inN) {
    n   = inN;
    stx = 1.0;
  }
  This(int inN, double sty) {
    this(inN);  // 上のコンストラクタを呼び出す。
    this.sty = sty;
  }
  void printField() {
    System.out.println("n = " + n + " stx = " + stx + " sty = " + sty);
  }
}
```

15　クラスの継承 (p.388)

- もとになるクラスがあり，そのクラスに，
  - 追加
  - 変更

  することによって，新しいクラスを作る．
- コピーしてから書き変え，類似したクラスを多数作ると．
  - 類似したクラスが多数できて，管理が困難．
  - もとのクラスを変更しても，新しいクラスは変更されない．
- **追加・変更**した部分だけを記述したい．
  - **スーパークラス**，親クラス ： 元になるクラス
  - **サブクラス**，子クラス： 派生して作られたクラス

15.1　extends (p.390)

- クラスを追加するためには，`extends`キーワードを使う．

```
class サブクラス名 extends スーパークラス名 {
  フィールドの宣言
  メソッドの定義
}
```

15.2 例

```
public class PartsBasic {
  int     id;        // 部品番号
  int     type;      // 部品タイプ
  String  name;      // 部品名
  int     price;     // 単価(円)
  int     period;    // 納期(日)
  // 引数のないコンストラクタ
  PartsBasic() {}
  // 引数を5つとるコンストラクタ
  PartsBasic(int id, int type, String name, int price, int period) {
    this.id     = id;                  this.type   = type;
    this.name   = new String(name);  this.price  = price;
    this.period = period;
  }
  void printInf() {  // 部品情報(部品名，業者名，単価，納期)を表示する。
    System.out.println("部品名：" + name + "type：" + type
        + "  単価：" + price + "円  納期：" + period + "日");
  }
}
```

```
public class ResistorBasic extends PartsBasic {
    double resistance;
    double maxPower;

    // 4つの引数をとるコンストラクタ
    ResistorBasic(int id, String name, int price, int period) {
      this.id     = id;
      this.type   = 1;
      this.name   = new String(name);
      this.price  = price;
      this.period = period;
    }
    // 抵抗器のスペックを表示
    void printSpec() {
      System.out.println("resistance = " + resistance
          + " maxPower = " + maxPower);
    }
}
```

メインプログラム

```
public class MainPartsSuper {
  public static void main(String[] args) {
    ResistorBasic   re = new ResistorBasic(1, "金属皮膜抵抗器", 45, 5);
    CapacitorBasic  cp = new CapacitorBasic(2, "セラミックコンデンサ", 10, 3);
    re.price = 35;
    re.printInf(); cp.printInf();
    re.resistance  = 1000;   re.maxPower   = 0.25;
    cp.capacitance = 1.0e-6; cp.maxVoltage = 16;
    re.printSpec(); cp.printSpec();
  }
}
```

結果

部品名：金属皮膜抵抗器　単価：35円　納期：5日
部品名：セラミックコンデンサ　単価：10円　納期：3日
resistance = 1000.0 maxPower = 0.25
capactance = 1.0E-6 maxVoltage = 16.0

15.3 super (p.393)

- superで，スーパークラスを表わす.
- スーパークラスのコンストラクタを呼び出すときなどに使う.

```
public class ResistorSuper extends PartsBasic {
  double resistance;
  double maxPower;

  // 4つの引数をとるコンストラクタ
  ResistorSuper(int id, String name, int price, int period) {
     super(id, 1, name, price, period);
  }
  // 抵抗器のスペックを表示
  void printSpec() {
    System.out.println("resistance = " + resistance
        + " maxPower = " + maxPower);
  }
}
```

15.4　メソッドのオーバーライド (p.406)

- サブクラスにおいて，スーパークラスと同じ
  - 識別子
  - 引数の型の並び
  - 戻り値の型

  を持ったメソッドを定義すること.
- オーバーライドによって，機能の追加でなく**変更**を行うことができる.
- メソッドを区別している識別子と引数の型の並びを，**シグネチャー**と呼ぶ.
- シグネチャーが同じで戻り値の型が異るメソッドを定義することは不可.
  (戻り値の型は，メソッドの区別に使っていない．)
- スーパークラスのオブジェクトを**参照する**変数は，スーパークラスのメソッドを呼び出す.
- サブクラスのオブジェクトを**参照する**変数は，サブクラスのメソッドを呼び出す.
- 「参照する」は**型ではない**(ポリモーフィズムを参照).

```
public class Parts {
  int     id;       // 部品番号
  String  name;     // 部品名
  int     price;    // 単価
  int     period;   // 納期
  // 引数を5つとるコンストラクタ
  Parts(int id, int type, int String name, int price, int period) {
    this.id     = id;
    this.type   = type;
    this.name   = new String(name);
    this.price  = price;
    this.period = period;
  }
  void printInf() { // 部品情報(部品名，業者名，単価，納期)表示メソッド
    System.out.println("部品名：" + name + "  単価：" + price
        + "円  納期：" + period + "日");
  }
  double calPower(double current, double voltage) {// 電力計算メソッド
    return current * voltage;
  }
```

```
}

public class Resistor extends Parts {
  double resistance = 1.0;
  double maxPower   = 1.0;

  // 引数を4つとるコンストラクタ
  Resistor(int id, String name, int price, int period) {
    super(id, 1, name, price, period);
  }
  // スペックの設定
  void setSpec(double resistance, double maxPower) {
    this.resistance = resistance;
    this.maxPower   = maxPower;
  }
  // 部品情報(部品名，業者名，単価，納期)と抵抗器のスペックを表示する。
  void printInf() {
    System.out.println("部品名：" + name + "  単価：" + price
        + "円  納期：" + period + "日");
    printSpec();
  }
```

```
  // 抵抗器のスペックを表示
  void printSpec() {
    System.out.println("resistance = " + resistance
        + " maxPower = " + maxPower);
  }
  // 消費電力を表示し，規格をオーバーしているようならば，表示する。
  double calPower(double current, double voltage) {
    double power = current * voltage;
    if (power > maxPower) {
      System.out.println("Power excess its limit(" + maxPower + "): "
                         + power);
      return -1.0;
    }
    return power;
  }
}
```

メインプログラム

```
public class MainParts {
  public static void main(String[] args) {
    Parts    pt = new Parts(0, 1, "部品", 35, 2);
    Resistor re = new Resistor(1, "金属皮膜抵抗器", 45, 5);
    re.setSpec(1.5, 0.25);
    pt.printInf(); re.printInf();
    System.out.println("Power of pt = " + pt.calPower(2.0, 3.0));
    System.out.println("Power of re = " + re.calPower(2.0, 3.0));
  }
}
```

出力

```
部品名：部品　単価：35円　納期：2日
部品名：金属皮膜抵抗器　単価：45円　納期：5日
resistance = 1.5 maxPower = 0.25
Power of pt = 6.0
Power excess its limit(0.25): 6.0
Power of re = -1.0
```

オブジェクトに対応してメソッドが呼び出されている。

15.5　ポリモーフィズム (p.408)

- 子クラスのオブジェクトは，**親クラスの型を持った変数に代入することができる．**
- このとき，オーバーライドしたメソッドを呼ぶと，**変数の型に係わらず，その変数が指しているオブジェクトのクラス**に応じたメソッドが呼ばれる．

```
public class MainPolymorphism {
  public static void main(String[] args) {
    Parts    pt1 = new Parts(0, 1, "部品", 35, 2);
    Parts    pt2;
    Resistor re  = new Resistor(1, "金属皮膜抵抗器", 45, 5);
    Parts    pt3 = new Resistor(1, "カーボン抵抗器", 15, 1);
    re.setSpec(1.5, 0.5);
    pt2 = re;
    pt1.printInf();
    pt2.printInf();
    re.printInf();
    pt3.printInf();
}
```

出力

```
部品名：部品　単価：35円　納期：2日
部品名：金属皮膜抵抗器　単価：45円　納期：5日
resistance = 1.5 maxPower = 0.5
部品名：金属皮膜抵抗器　単価：45円　納期：5日
resistance = 1.5 maxPower = 0.5
部品名：カーボン抵抗器　単価：15円　納期：1日
resistance = 1.0 maxPower = 1.0
```

型でなく参照しているオブジェクトに応じて，メソッドが呼び出されている。

15.6　参照型のキャスト (p.412)

- ポリモーフィズムにおいて，変数の型がスーパークラスの型であると，**子クラスにしかない，フィールドやメソッドは参照できない．**
- 前の例で，次の記述はコンパイルエラーを引き起こす。
  - `pt3.setSpec(10.0, 2.0);`
  - `pt3.printSpec();`
- キャストによって，その変数が参照しているものが，子クラスのオブジェクトであることを明示する。
- **スーパークラスの型の変数を，サブクラスの型にキャストすることができる**。実行時にその変数がそのサブクラスのオブジェクトを参照していないと，キャストの実行時にエラーが発生する。
- 先の例では,
  - `((Resistor) pt3).setSpec(10.0, 2.0);`
  - `((Resistor) pt3).printSpec();`

  とすれば，コンパイルエラーが生じない.
- **注意**：スーパクラスのオブジェクト参照型変数から，サブクラスの参照型変数へ直接代入することはできない(コンパイルエラーになる。キャストすれば良い)。

15.7　クラス型変数配列

- クラス型変数はオブジェクトへの参照情報を格納している。
- 例：クラスStudentSortが定義されているとき，

```
StudentSort studentL[];
```

は，クラスStudentSortのオブジェクトの参照情報を格納する配列変数studentLを宣言している。

- 実際には，studentLは，StudentSortの**オブジェクトへの複数の参照情報を格納している領域への参照情報を格納する。**
- そして，

```
studentL = new StudentSort[10];
```

によって，クラスStudentSortのオブジェクトの参照情報を格納する10個分の領域が確保され，その領域の参照情報がstudentLに格納される。

- まとめて，以下のように書くこともできる。

```
StudentSort studentL[] = new StudentSort[10];
```

- この文だけでは，**オブジェクトは1つもできていない。**実際にオブジェクトを使うためには，オブジェクトを作成して，その参照情報を配列に格納する必要がある。

- クラスStudentSort()のコンストラクタStudentSort() を使うならば, 以下のようにする。

```
StudentSort studentL[] = new StudentSort[10];
studentL[0] = new StudentSort();
studentL[1] = new StudentSort();
studentL[2] = new StudentSort();
studentL[3] = new StudentSort();
studentL[4] = new StudentSort();
studentL[5] = new StudentSort();
studentL[6] = new StudentSort();
studentL[7] = new StudentSort();
studentL[8] = new StudentSort();
studentL[9] = new StudentSort();
```

もちろん, for文などを使っても良い。

```
StudentSort studentL[] = new StudentSort[10];
for(int k = 0 ; k < 10 ; ++k) studentL[k] = new StudentSort();
```

15.8　その他

(本講義では，説明は省略する)

15.8.1　抽象クラス (p.426)

- 抽象クラスはオブジェクトにはなれない。
- 抽象クラスを継承したクラスがオブジェクトになれる。
- キーワードabstractを用いる.

  ```
  abstruct class Parts {
  ```

- メソッドにabstractを使うことができる。その場合，メソッドの内容は記述しない。

  ```
  abstruct double calPower(double current, double voltage);
  ```

- サブクラスでそのメソッドを定義する (定義しないとコンパイルエラーになる)。
- 通常のクラスでは，抽象メソッドを定義できない。
- 抽象クラスに普通のフィールドやメソッドを書いても良い。

15.8.2　インターフェース (p.454)

- Javaでは，親とすることができるクラスは１つだけ。
- 複数の機能を使うために，多数のクラスを継承できた方が便利であるが，プログラムがわかりにくくなるなど弊害も多い。
- インターフェース：それを継承したクラスを，外部のプログラムがどのように扱うか規定するものである。
- 抽象クラスに近いものである。
- **複数のインターフェースを継承することができる。**
- `interface`を使う。
- フィールドはクラス変数で`final`でないといけない(つまり定数)。
- メソッドは`abstract`でないといけない。
- 例

```
interface CurrentIntf {
  static final double coeffc = Math.sqrt(0.5);
  abstract double getEffCurrent();
  abstract void setPeakCurrent(double peakCurrent);
}
```

15.8.3　パッケージ (p.364)

- クラス名などが重なると問題になる。
- **名前空間** (名前が通用する範囲) を，パッケージによって分割する。
- `java.awt.event`
  は，`java`という大きなパッケージに含まれる`awt`というパッケージに含まれる。`event`というパッケージを意味する。
- このパッケージのクラス`ActionEvent`を利用するときは，プログラムのはじめの方に，次のように記述する。

  ```
  import java.awt.event.ActionEvent
  ```
- そのパッケージのクラスをすべて利用できるようにするためには，プログラムのはじめの方に，次のように記述する。

  ```
  import java.awt.event.*
  ```
- パッケージ外のプログラムから使用されるクラスは，`public`で宣言する必要がある。
- プログラムのパッケージを指定するときは，`package`文で指定する。
- クラスファイルを置くフォルダの名前は，パッケージの名前と一致していなくてはいけない。
- `System.out.println()`のクラス`System`は，パッケージ`java.lang`に属するが，このパッケージに関しては，`import`文を省略できる。

16　クラスに関する演習

ベクトル計算のためのクラスを作成する。

- 名前：Vector
- フィールド
  - name：String型のフィールドで名前を格納する。
  - dimension：int型のフィールドで，ベクトルの次元を格納する。
  - component：double型の配列のフィールドで，ベクトルの各成分を格納する。
- メソッド
  - Vector(String name, int dimension)：コンストラクタで，名前と次元を引数として，それぞれをフィールドに格納する。そして，次元分のdoubleの領域を確保し，その参照情報をcomponentに格納する。
  - setComponent(int l, double comp)：自身のl番目(0番目が先頭)の成分(compnent[l])にcompを格納する。
  - void addVect(Vector x, Vector y)：引数の2つのベクトルを加算し自身に格納する。
  - double innerProduct(Vector x)：自身のベクトルと引数のベクトルの内積を計算して，戻り値とする。
  - void printVector()：自身のベクトルを表示する。

例えば,

$$x = \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix}, \qquad y = \begin{pmatrix} 3 \\ 2 \\ -2 \end{pmatrix},$$

とし，$z = x + y$を計算し，x, y, zを表示した後，xとyの内積を計算した結果は，次のようになる。

```
x[0] = 2.0
x[1] = -1.0
x[2] = 1.0
y[0] = 3.0
y[1] = 2.0
y[2] = -2.0
z[0] = 5.0
z[1] = 1.0
z[2] = -1.0
<x,y> = 2.0
```

そのメインプログラムを，次ページに記す。

```
public class MainVector {
  static public void main(String args[]) {
    Vector x = new Vector("x", 3);
    Vector y = new Vector("y", 3);
    Vector z = new Vector("z", 3);
    x.setComponent(0, 2.0);
    x.setComponent(1, -1.0);
    x.setComponent(2, 1.0);
    y.setComponent(0, 3.0);
    y.setComponent(1, 2.0);
    y.setComponent(2, -2.0);
    z.addVect(x, y);
    x.printVector();
    y.printVector();
    z.printVector();
    System.out.println("<x,y> = " + x.innerProduct(y));
  }
}
```

16.1　クラス型変数の配列によるソート

- 1人分の学生のデータをまとめて扱うことができる。
- **並べる順番の決め方を，学生のクラスで決めることができる。**StudentSortのクラスメソッド　boolean comp(int type, StudentSort x, StudentSort y)が，trueならば，xの方がyよりも先に並ぶ。

```
class StudentSort {
  private int stdNo, pMath, pPhys;
  private String name;
  private double height, aveP;

  StudentSort(int stdNo, String name, double height) {
    this.stdNo  = stdNo;
    this.name   = name;
    this.height = height;
  }
  void setPoint(int inPMath, int inPPhys) {
    pMath = inPMath;
    pPhys = inPPhys;
    setAvePoint();
  }
  void setAvePoint() {
    aveP = (pMath + pPhys) * 0.5;
  }
```

```
  static boolean comp(int type, StudentSort x, StudentSort y) {
    boolean res = false;
    switch (type) {
    case  0: res = x.stdNo > y.stdNo; break;
    case 10: res = x.stdNo < y.stdNo; break;
    case  1: res = x.pMath > y.pMath; break;
    case 11: res = x.pMath < y.pMath; break;
    case  2: res = x.pPhys > y.pPhys; break;
    case 12: res = x.pPhys < y.pPhys; break;
    case  3: res = x.aveP  > y.aveP ; break;
    case 13: res = x.aveP  < y.aveP ; break;
    case  4: res = x.height > y.height; break;
    case 14: res = x.height < y.height; break;
    }
    return res;
  }
  void print() {
    System.out.printf("      %4d  %6s:%4d %4d %6.1f %6.1f \n",
                      stdNo, name, pMath, pPhys, aveP, height);
} }
```

```
public class ClassSort {
  private int nStudents;
  private StudentSort[] studentL;
  // コンストラクタ
  ClassSort(int nStudents) {
    this.nStudents = nStudents;
    studentL = new StudentSort[nStudents];
  }
  // 配列のno番に学生のデータをセットする
  void setStudent(int no, int stdNo, String name, int mathP, int physP,
                  double height) {
    studentL[no] = new StudentSort(stdNo, name, height);
    studentL[no].setPoint(mathP, physP);
  }
```

```
  void sortSelect(int type) {

    // typeによって並べ方が変わるソートの部分を考えよう！









  }

  void print() {
    System.out.println(
           "配列番号  学籍番号      名  前    数学  物理  平均    身長");
    for (int k = 0 ; k < nStudents ; ++k) {
      System.out.printf("   %4d", k); studentL[k].print();
    }
  }
}
```

```
public class MainClassSort {
  public static void main(String args[]) {
    ClassSort cls = new ClassSort(6);
    cls.setStudent(0, 0, "内山昂輝　", 95, 95, 176.0);
    cls.setStudent(1, 1, "日笠陽子　", 89, 89, 157.0);
    cls.setStudent(2, 2, "野上ゆかな", 90, 82, 160.0);
    cls.setStudent(3, 3, "下田麻美　", 88, 85, 158.0);
    cls.setStudent(4, 4, "花澤香菜　", 80, 87, 156.7);
    cls.setStudent(5, 5, "井上麻里奈", 91, 85, 162.0);

    cls.sortSelect(4);

    cls.print();
  }
}
```

16.2　複素数の絶対値を使ってソート

先の問題は少し難しすぎたため，もう少し簡単な例。
複素数のクラス

```
public class ComplexNum {
  protected double re, im; // re：実部，im：虚部
  // 実部と虚部の値を引数とするコンストラクタ
  ComplexNum(double re, double im) {
    this.re = re;
    this.im = im;
  }
  // 自身の絶対値を返すメソッド
  double abs() {
    return Math.sqrt(re * re + im * im);
  }
  // 自身の実部と虚部を表示するメソッド
  void print() {
    System.out.println(re + " + " + im +"i");
  }
}
```

絶対値に従って降順に選択ソートでソートするメインプログラムを完成させよう。

```
public class MainSortComplex {
  public static void main(String args[]) {
    int N = 10;
    ComplexNum cnL[] = new ComplexNum[N];
    cnL[0] = new ComplexNum( 1.0,  1.0);
    cnL[1] = new ComplexNum( 3.0,  1.0);
    cnL[2] = new ComplexNum( 1.0, -4.0);
    cnL[3] = new ComplexNum( 2.0, -2.0);
    cnL[4] = new ComplexNum( 4.0,  0.0);
    cnL[5] = new ComplexNum( 0.0, -2.0);
    cnL[6] = new ComplexNum(-1.0, 3.0);
    cnL[7] = new ComplexNum(-2.0, 3.0);
    cnL[8] = new ComplexNum( 3.0,  4.0);
    cnL[9] = new ComplexNum( 2.0, -1.0);

    // 入力データを表示
    for (int i = 0 ; i < N ; ++i) cnL[i].print();
```

```
    // 選択ソートを実施

    // 結果を表示
    for (int i = 0 ; i < N ; ++i) cnL[i].print();
  }
}
```

16.3　多数桁の四則演算

```
// 多数の桁の整数の四則演算を行うクラス
// 配列の1つに4桁の整数を格納する。
public class MInt {
  int N;          // 桁数 / 4
  boolean sign = false; // false : +, true : -
  int val[];
  MInt(int N) {
    this.N = N;
    val = new int[N];
    clear();
  }
  // 値を表示
  void print(String s) {
    if (sign) System.out.print(s + "-");
    else System.out.print(s + "+");
    for (int k = N - 1 ; k >= 0 ; --k) System.out.printf("%04d", val[k]);
    System.out.println();
  }
```

```
// 値をクリアする
void clear() {
  sign = false;
  for (int k = 0 ; k < N ; ++k) val[k] = 0;
}
// xを自分にコピーする
void copy(MInt x) {
  sign = x.sign;
  for (int k = 0 ; k < N ; ++k) val[k] = x.val[k];
}
// 絶対値でxの方が大きい場合 true
boolean gtAb(MInt x, MInt y) {
  for (int k = N - 1 ; k >= 0 ; --k) {
    if (x.val[k] > y.val[k]) return true;
    else if (x.val[k] < y.val[k]) return false;
  }
  return false;
}
```

```
// 符号付きでxがyよりも大きいときにtrue
boolean gt(MInt x, MInt y) {
  if ((! x.sign) && y.sign) return true;         // 正と負
  else if (x.sign && (! y.sign)) return false; // 負と正
  else if (x.sign && y.sign) return gtAb(y, x); // 負と負
  else return gtAb(x, y); // 正と正
}
// xとyの絶対値の和を自分に格納する
void addAb(MInt x, MInt y) {
  int carry = 0;
  for (int k = 0 ; k < N ; ++k) {
    int tmp = x.val[k] + y.val[k] + carry;
    if (tmp > 9999) {
      val[k] = tmp - 10000;
      carry  = 1;
    } else {
      val[k] = tmp;
      carry  = 0;
    }
} }
```

```
// xとyの絶対値の差を自分に格納する(|x! > !y!を仮定)
void subAb(MInt x, MInt y) {
  int borrow = 0;
  for (int k = 0 ; k < N ; ++k) {
    int tmp = x.val[k] - y.val[k] - borrow;
    if (tmp < 0) {
    val[k] = tmp + 10000;
    borrow = 1;
    } else {
      val[k] = tmp;
      borrow = 0;
    }
  }
}
```

```
// 符号付きの和
void add(MInt x, MInt y) {
  if (x.sign ^ y.sign) {
    if (gtAb(x, y)) {
      subAb(x, y);
      sign = x.sign;
    } else {
      subAb(y, x);
      sign = y.sign;
    }
  } else {
    addAb(x, y);
    sign = x.sign;
  }
}
```

```
// 符号付きの差
void sub(MInt x, MInt y) {
  if (x.sign ^ y.sign) {
    addAb(x, y);
    sign = x.sign;
  } else {
    if (gtAb(x, y)) {
      subAb(x, y);
      sign = x.sign;
    } else {
      subAb(y, x);
      sign = y.sign;
    }
  }
}
```

```
// xとyの積を自分に格納する。
void mul(MInt x, MInt y) {
  clear();
  sign = x.sign ^ y.sign;
  MInt tmp = new MInt(N);
  for (int k = 0 ; k < N ; ++k) {
    tmp.clear();
    tmp.mulS(x, y.val[k], k);
    addAb(this, tmp);
  }
}
// xと一つのint(4桁分)の積を自分に格納する。
void mulS(MInt x, int y, int k) {
  int carry = 0;
  for (int l = 0 ; l < N - k ; ++l) {
    int tmp = x.val[l] * y + carry;
    val[l + k] = tmp % 10000;
    carry      = tmp / 10000;
  }
}
```

```
// xをyで割った商を自分に格納する。
void div(MInt x, MInt y) {
  clear();
  sign = x.sign ^ y.sign;
  int xTop, yTop;
  for (xTop = N - 1 ; xTop >= 0 ; --xTop) {
    if (x.val[xTop] != 0) break;
  }
  if (xTop < 0) return; // 結果は0
  for (yTop = N - 1 ; yTop >= 0 ; --yTop) {
    if (y.val[yTop] != 0) break;
  }
  if (yTop < 0) return; // 0割

  long tApp, yApp; // 近似値
  yApp = y.val[yTop] * 10000L;
  if (yTop >= 1) yApp += y.val[yTop - 1];
```

```
MInt tmp1 = new MInt(N);
MInt tmp2 = new MInt(N);
tmp1.copy(x);

for (int l = xTop - yTop ; l >= 0 ; --l) {
  tApp = tmp1.val[yTop + l] * 10000L;
  if (yTop + l < N - 1) {
    tApp += tmp1.val[yTop + l + 1] * 100000000L;
  }
  if (yTop + l > 0) tApp += tmp1.val[yTop + l - 1];
  int quat = (int) (tApp / yApp); // 商の近似計算
  if (quat > 9999) quat = 9999;
  if (l <= 1) quat += 2;
  tmp2.clear();
  tmp2.mulS(y, quat, l);
  while(gtAb(tmp2, tmp1)) {   // 近似で求めた商が大きすぎた場合の修正
    --quat;
    tmp2.subAbS(tmp2, y, l);
  }
```

```
    tmp1.subAb(tmp1, tmp2);
    while(gtAbS(tmp1, y, l)) {   // 近似で求めた商が小さすぎたときの修正
      ++quat;
      tmp1.subAbS(tmp1, y, l);
    }
    val[l] = quat;
  }
}
// xの絶対値がyの絶対値を4桁で1個右シフトしたものよりも大きい
boolean gtAbS(MInt x, MInt y, int l) {
  int k;
  for (k = N - 1 ; k >= l ; --k) {
    if (x.val[k] > y.val[k - l]) return true;
    else if (x.val[k] < y.val[k - l]) return false;;
  }
  for (; k >= 0 ; --k) {
    if (x.val[k] > 0) return true;
  }
  return false;
}
```

```
  // xの絶対値からyの絶対値を4桁で1個だけ右シフトしたものとの差
  void subAbS(MInt x, MInt y, int l) { // 絶対値の差 |x! > !シフトした y!
を仮定
    int borrow = 0, k;
    for (k = 0 ; k < l ; ++k) {
      val[k] = x.val[k];
    }
    for ( ; k < N ; ++k) {
      int tmp = x.val[k] - y.val[k - l] - borrow;
      if (tmp < 0) {
      val[k] = tmp + 10000;
      borrow = 1;
      } else {
        val[k] = tmp;
        borrow = 0;
      }
    }
  }
}
```

```
public class MainMInt {
  public static void main(String args[]){
    MInt x = new MInt(4);
    MInt y = new MInt(4);
    MInt z = new MInt(4);
/*
    x.sign = true;
    x.val[1] = 8234; x.val[0] = 3234;
    y.val[1] = 4321; y.val[0] = 8132;
    x.print("x = "); y.print("y = ");
    z.mul(x, y);
    z.print("z = ");
    x.clear(); y.clear(); z.clear();

    x.val[2] = 345; x.val[1] = 8234; x.val[0] = 3234;
    y.val[1] = 4321; y.val[0] = 8132;
    x.print("x = "); y.print("y = ");
    z.div(x, y);
    z.print("z = ");
*/
```

```
x.clear(); y.clear();
x.val[3] = 3256; x.val[2] = 345; x.val[1] = 8234; x.val[0] = 3234;
y.val[1] = 4321; y.val[0] = 8132;
x.print("x = "); y.print("y = ");
z.div(x, y);
z.print("z = ");

// 自然対数の計算
int N = 26;
MInt one = new MInt(N);
MInt exp = new MInt(N);
MInt oneL = new MInt(N);
MInt n = new MInt(N);
MInt tmp = new MInt(N);

one.val[0] = 1;
oneL.val[N - 1] = 1;
exp.val[N - 1] = 1;
n.val[0] = 100;
```

```
    for (int k = 100 ; k >= 1; --k) {
      tmp.div(exp, n);
      exp.add(oneL, tmp);
      tmp.sub(n, one);
      n.copy(tmp);
    }
    exp.print("exp = ");
  }
}
```

16.4　まとめ

- クラス，オブジェクト
- フィールド，メソッド，ローカル変数
- コンストラクタ
- オーバーロード
- 継承は，システムを構造化し，過去のプログラム資産を有効に利用するために重要
- クラス型の配列変数の領域を確保しても，オブジェクトの領域は取られないことに注意すること。