ドゥビドゥバ、ジャバ
ー直感Javaのオブジェクトー

岐阜経済大学　経営学部 経営情報学科　井戸　伸彦

来歴：

０．０版　２００４年８月１８日

スライドの構成

はじめに

（１）このスライドで学ぶこと

（２）クラスを作る

（３）アクセスの可否

（４）コンストラクタ

（５）クラス変数とクラスメソッド

（６）クラス型変数

（７）パッケージ

（８）標準クラス

（９）基本的な標準クラスの利用

（１０）継承

（１１）インタフェース

はじめに

■本スライドは、Javaによるプログラミングを説明するために使用するスライドです。単独で教科書となることは意図していません。

■本スライドでは、Javaの限られた機能についてのみ扱っています。次のスライドの続編として作成されています。

- 「シャバドゥビ、ジャバ　ー速習Java覚書ー」

（http://www.gifu-keizai.ac.jp/~ido/doc/java/java_text.pdf）

■Javaについてきちんと勉強したい方は、プログラミング等の講義を受講して頂くことをお願いします。

■説明は直感的な分かりやすさを重視し、厳密には不正確な言い方もしています。

■Javaの実行・デバッグ環境としては、eclipseを前提としてます。これについては、下記の資料を参考にしてください。

- 月に吠える　ーeclipseによるJavaアプリケーション作成ー

（http://www.gifu-keizai.ac.jp/~ido/doc/java/eclipse_application.pdf）

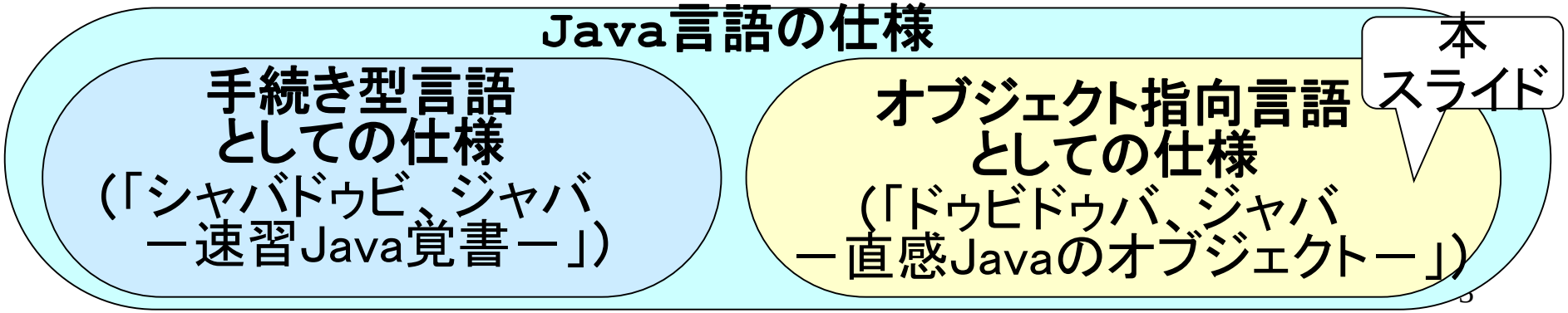
（1）このスライドで学ぶこと

■スライド「シャバドゥビ、ジャバ　ー速習Java覚書ー」でJavaについて勉強しました。

- そこで扱った内容は、C言語とあまり違わないもので、“Javaらしい部分”は扱っていませんでした。
- C言語以外の多くの言語でも、書き方が多少異なるだけで、if文やfor文の“考え方”は皆同じです。これらの言語は考え方において共通である、すなわち、「これらは手続き型言語である」と言います。

■本スライドでは、“Javaらしい部分”を扱っていきます。

- Javaはオブジェクト指向言語です。本スライドで扱う“Javaらしい部分”は、主にオブジェクト指向に由来するものです。



(2)クラスを作る

■サッカー選手の名前とゴール数とを管理する、“player”というクラスを考えてみます。

```
public class FootballTeam {
  public static void main(String[] args) {
    Player pl;
    pl = new Player();
    pl.name="高原";
    pl.goals=3;
    pl.show();
  }
}
```

<FootballTeam.java>

別ファイルとします。

```
public class Player {
  String name;
  int goals;
  void show(){
    System.out.println("名前："+name);
    System.out.println("得点："+goals);
  }
}
```

<Player.java>

（2．1）実行結果

■前スライドに記したクラス“FootballTeam”を起動すると、次のような実行結果になります。

<FootballTeam.java>

```
public class FootballTeam {
  public static void main(String[] args) {
    Player pl;
    pl = new Player();
    pl.name="高原";
    pl.goals=3;
    pl.show();
  }
}
```

<Player.java>

```
public class Player {
  String name;
  int goals;
  void show(){
    System.out.println("名前："+name);
    System.out.println("得点："+goals);
  }
}
```

＜実行結果＞

```
$ javac FootvallTeam.java
$ java FootballTeam
名前：高原
得点：3
```

（2．2）クラス“player”の構成

■クラス“**Player**”には、次の仕組みが備わっています。

```
public class Player {
  String name;
  int goals;

  void show(){
    System.out.println("名前："+name);
    System.out.println("得点："+goals);
  }
}
```

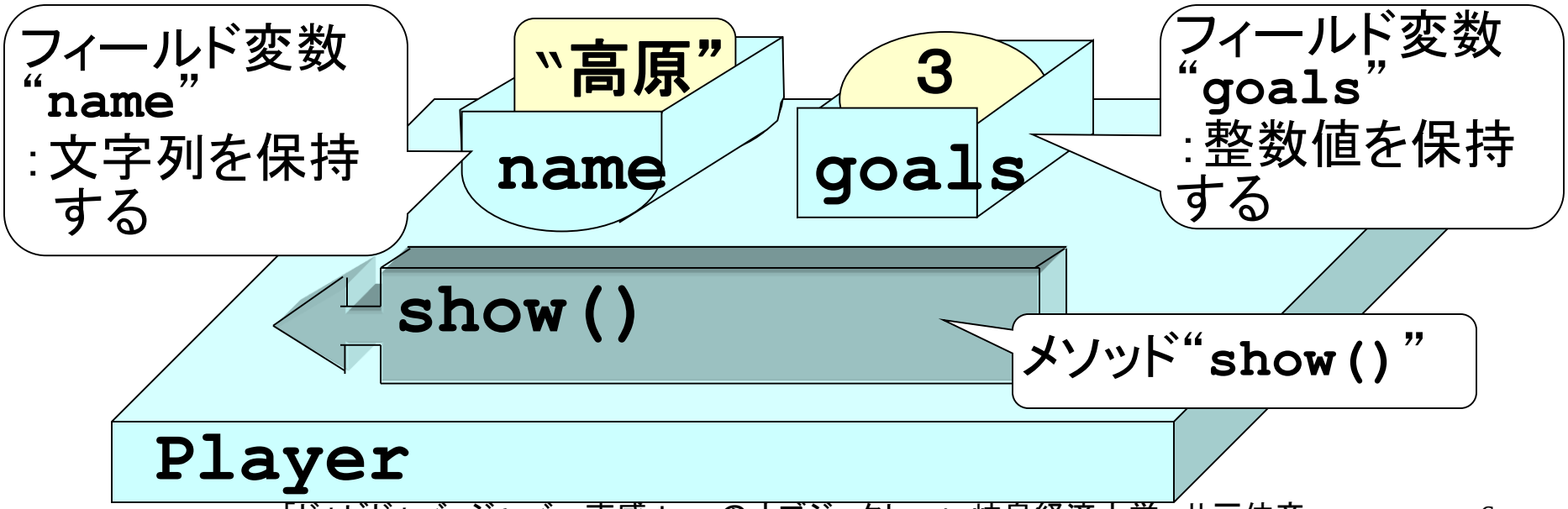
クラス

フィールド変数

メンバ

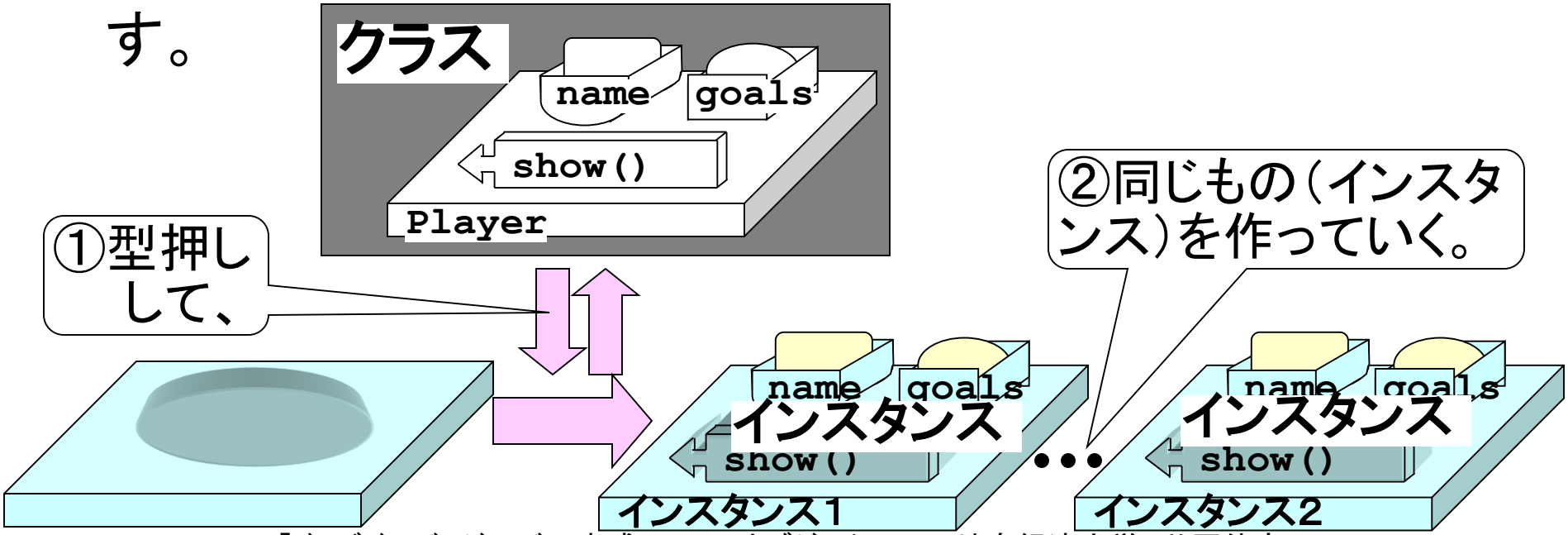
メソッド

フィールド変数とメソッドのこと



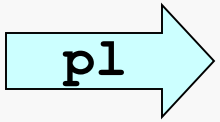
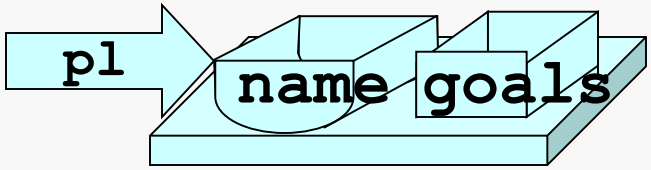
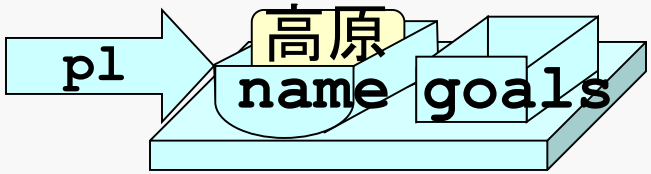
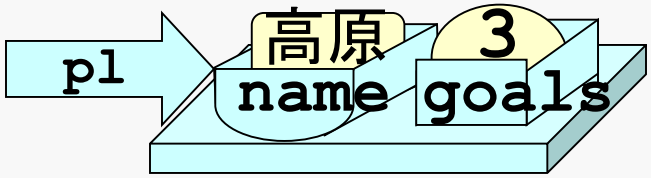
（2．3）クラスとインスタンス

- 前スライドに示したクラス“Player”の仕組みは、実際にこれを使う際、“インスタンス”と呼ばれるものを生成してから使用します。
- クラスは言わば“金型”みたいなもので、これにより“型押し”してインスタンスを生成し、これを用います。
- 同じクラスから、インスタンスはいくつでも生成できます。



（2．4）インスタンス化（生成）

■インスタンスの生成は、次のように行われます。

操作	記述	操作後の状態
インスタンスへのインデックス(クラス型の変数)を用意する	`Player pl;`	
インスタンスを生成し、インデックスに関連付ける	`pl=new Player();`	
フィール変数nameに“高原”を入れる。	`pl.name="高原";`	
フィールド変数goalsに“3”を入れる	`pl.goals=3;`	

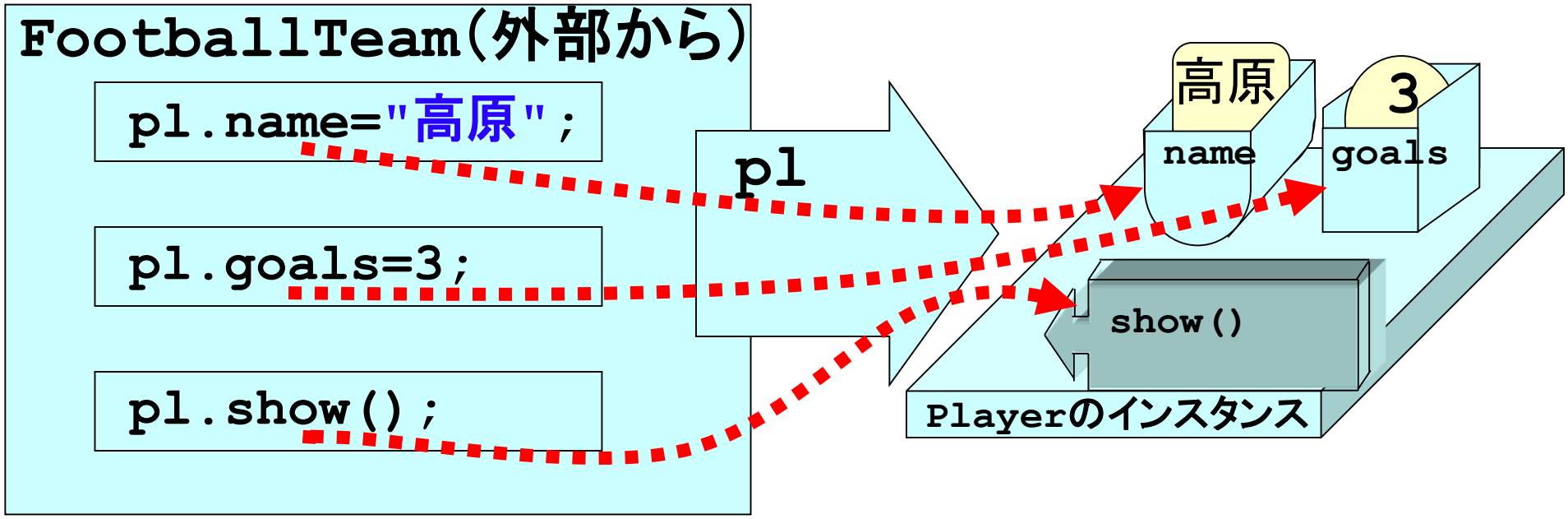
■最初の2行は`Player pl= new Player();`でもOKです。

（2．5）外からフィールド、メソッドを呼ぶ

■クラスの外部からフィールド／メソッドへのアクセスは、次の形で行います。

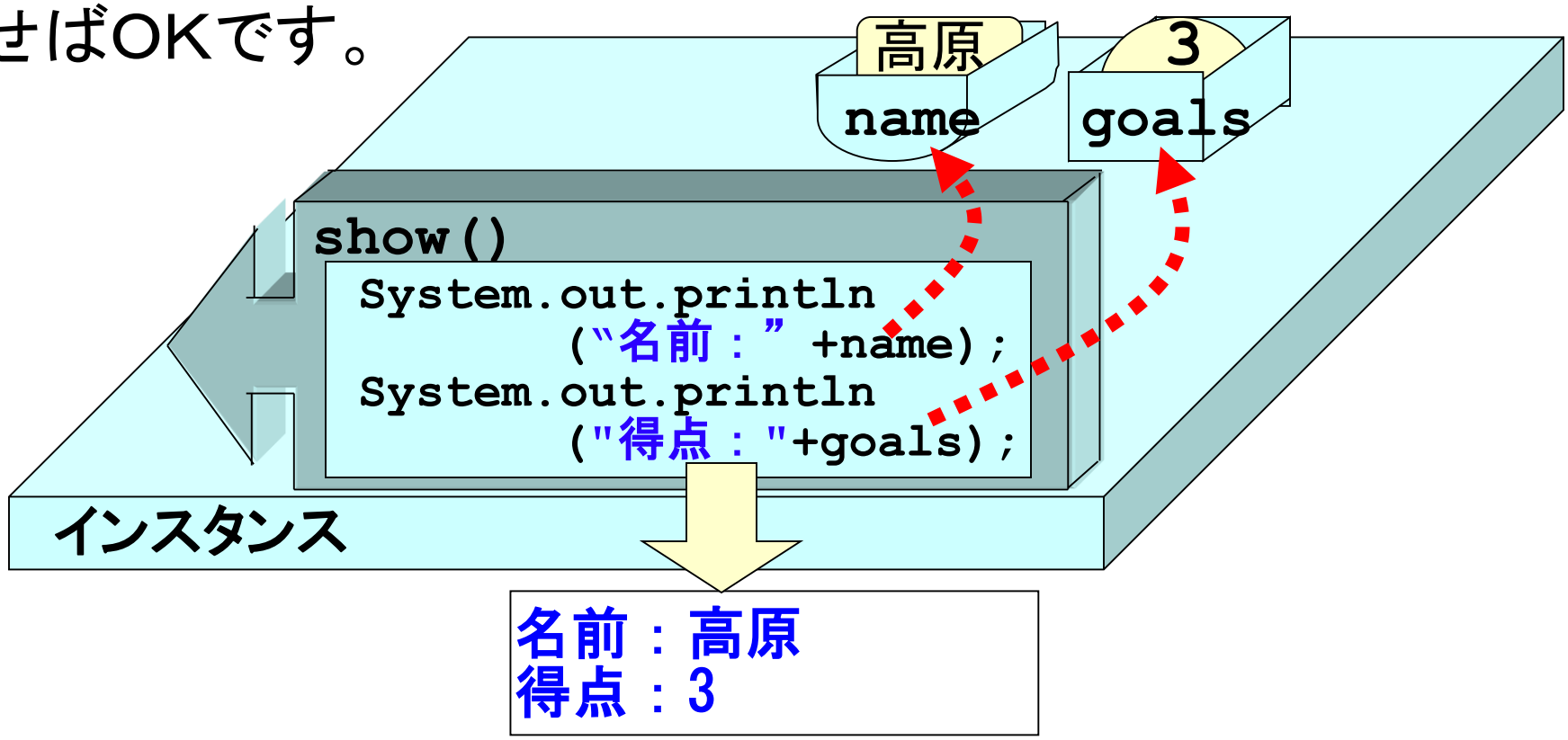
クラス型の変数.フィールド名

クラス型の変数.メソッド名



（2．6）内部のメソッドから

■クラスの内部のメソッドから、フィールド／メソッドにアクセスする際には、フィールド名／メソッド名のみを記せばOKです。



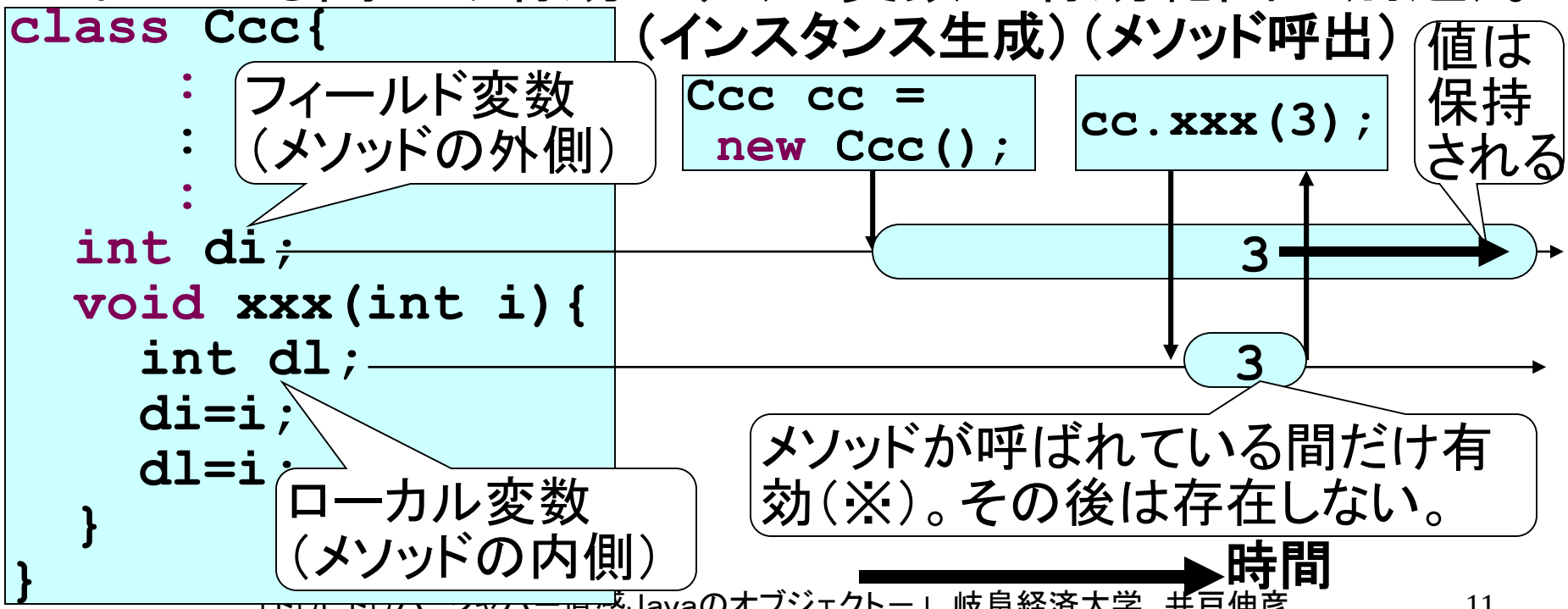
■自分自身のインスタンスを示す“this”を使って、

`this.name` のように記しても同じ意味になります。

（2．7）フィールド変数とローカル変数

■スライド「速習Java覚書」で作成してきたJavaのプログラムでは、変数はメソッドの内部で宣言してきました。このような変数をローカル変数と言います。

■フィールド変数はインスタンスが生きている間値を保持していますが、ローカル変数の値はメソッドが呼ばれている間だけ有効です（※変数の有効範囲は別途）。



（2．8）課題

■課題２－１
- スライド(2)のプログラムを作成してください。
- Playerインスタンスを2つ生成して、下記(A)のように表示するように、FootballTeamクラスを変更してください。

■課題２－２
- 次のような2つのクラスを作成してください。
  - ◆野球選手クラス BPlayer
    - □フィールドとして、名前(name)、守備位置(position)、得点(run)を持つ。
    - □上記のフィールドを出力するメソッドを持つ。
  - ◆野球チームクラス　BaseballTeam
    - □BPlayerクラスを3つ生成して、下記(B)の表示を行う。

```
$ java FootballTeam
名前：高原
得点：３
名前：久保
得点：４
```
(A)

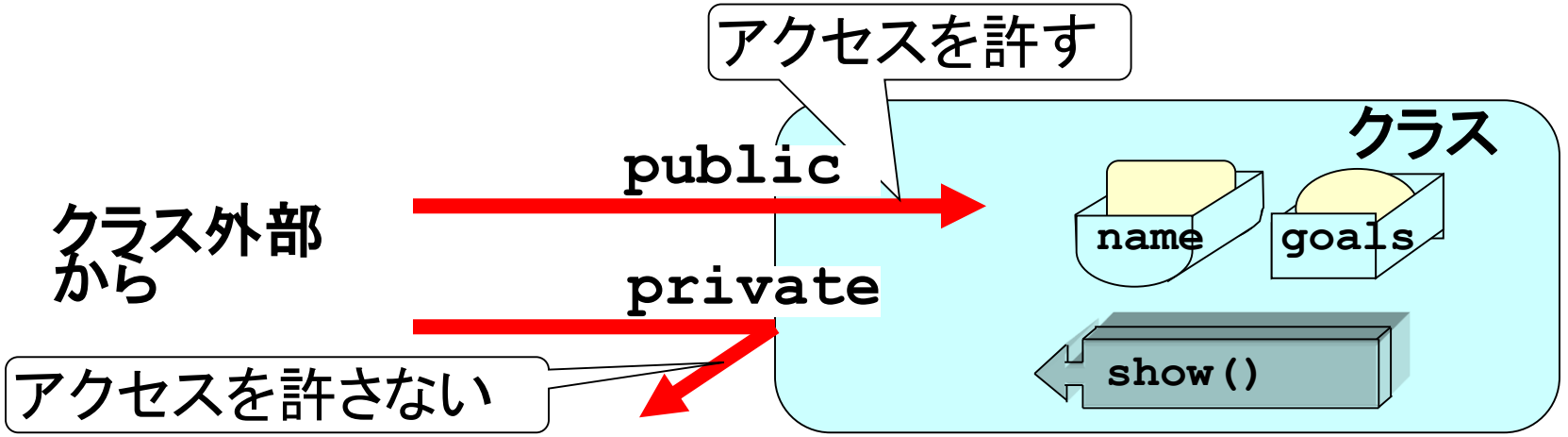
```
$ java BaseballTeam
イチロー　外野手　１３０
松井　　外野手　１２０
野茂　　投手　　５
```
(B)

（3）アクセスの可否、カプセル化

■クラスには、そのフィールド／メソッドに外部からのアクセスを許すか許さないかの可否を設定することが出来ます。外部からアクセス出来る範囲を制限することを、カプセル化と言います。

■アクセスの可否は、次のようなアクセス修飾子をフィールド／メソッドに付けることで指定します。

- **public**　：アクセスを許す
- **private**：アクセスを許さない



（3．1．1）どうして隠すのか？

■ここでは簡単な例を用いて、フィールドやメソッドを隠す（アクセスを許さない）ことの利点を見ていきます。

<FootballTeam2.java>

```
public class FootballTeam2 {
  public static void main(String[] args) {
    Player2 pl;
    pl = new Player2();
    pl.setGoals(-1);
    pl.show();
    pl.setGoals(3);
    pl.show();
  }
}
```

```
public class Player2 {
  private int goals;
  public void show(){
System.out.println("得点："+goals);
  }
  public void setGoals(int goals){
    if(goals<0){
      this.goals = 0;
    }else{
      this.goals = goals;
    }
  }
}
```

<Player2.java>

※このプログラムでは、フィールド“name”を省いています。

（３．１．２）好ましくない値の防止

■前スライドのプログラムを実行すると、次のような結果になります。

得点をマイナスの値に
設定しようとすると、“0”が設定される

<FootballTeam2.java>

```
pl.setGoals(-1);
pl.show();
pl.setGoals(3);
pl.show();
```

得点：0

得点：3

0以上の値は、普通に設定される

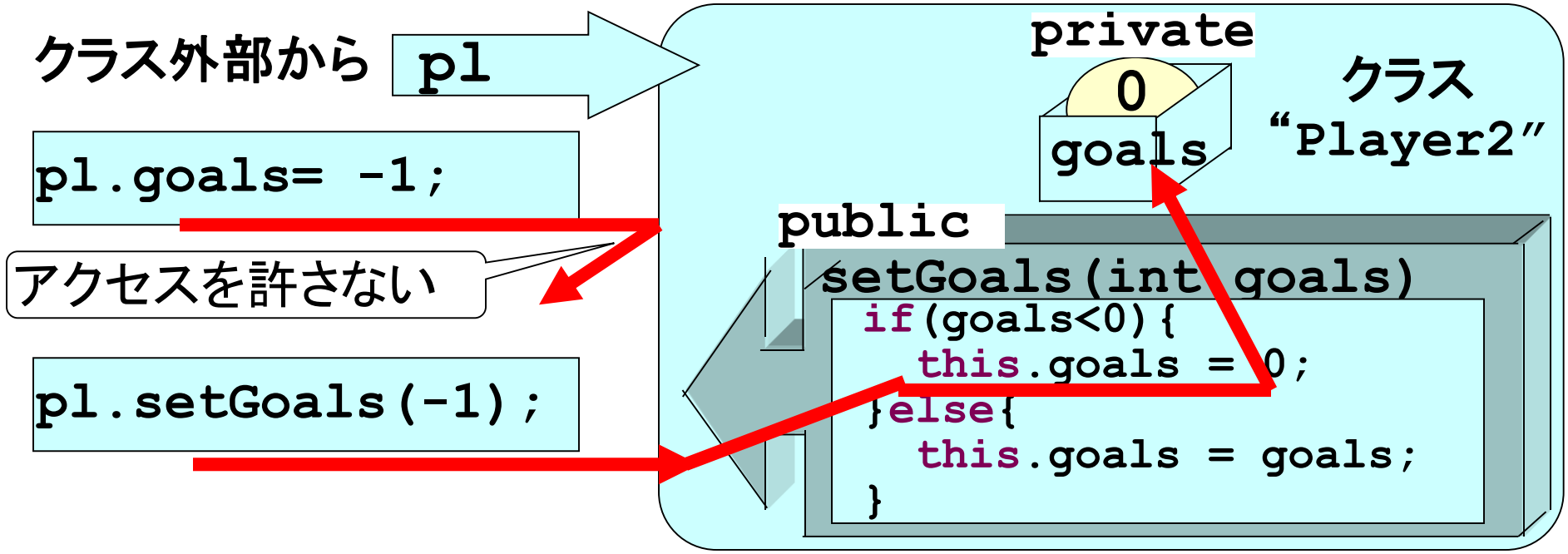
■得点（goals）の値がマイナスであることはおかしいので、そのようにならないようにしています。

■このように好ましくない値を防止すること出来るのは、次のスライドのような仕組みによります。

（３．１．３）データ保護のしくみ

■クラス“Player2”では、フィールド“goals”は“private”となっており、クラスの外から直接アクセス出来ません。

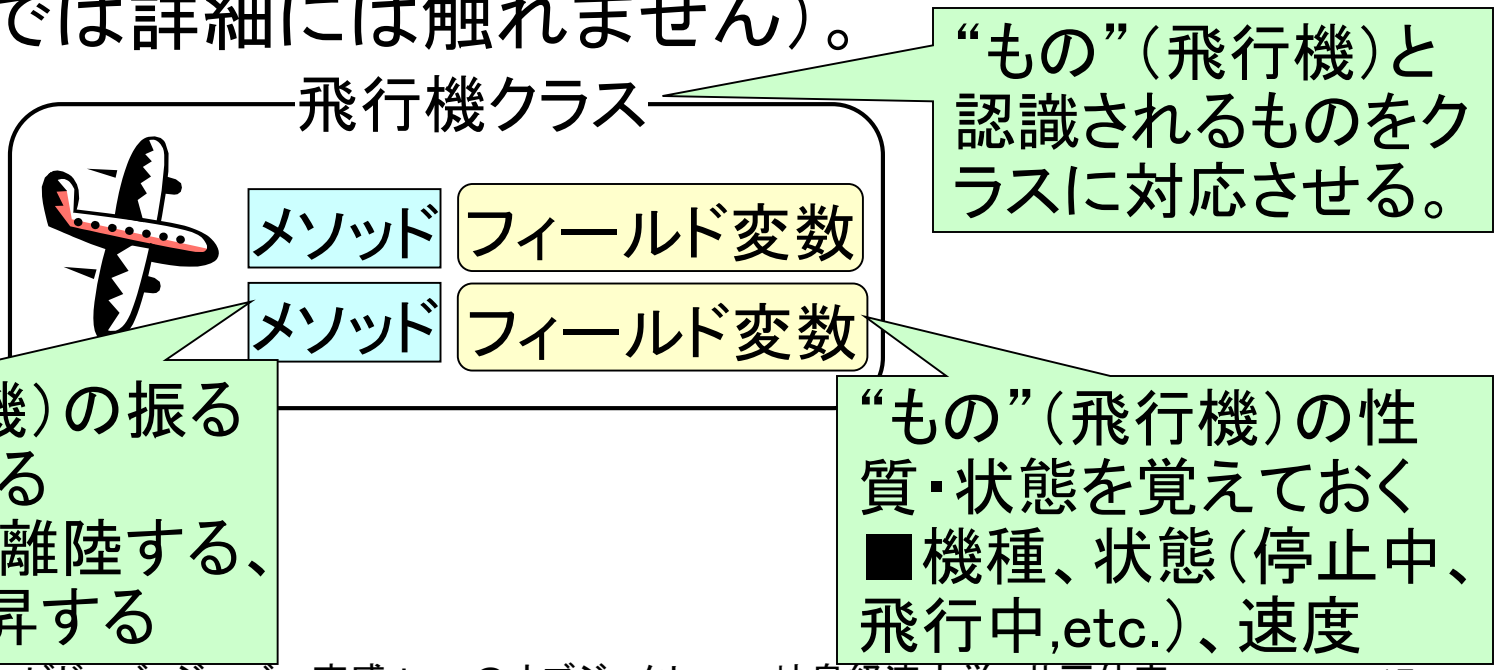
■替わりにメソッド“setGoals”を用いて“goals”を設定することが出来ます。このとき、マイナスの値が設定されようとすると、替わりに“0”を設定しています。



（３．２．１）クラスの考え方

■このスライドでは、「プログラミング言語としてのクラス」の機能について説明しています。プログラミング言語の規約には、「どのようなクラスを作れ」という内容がある訳ではありません。

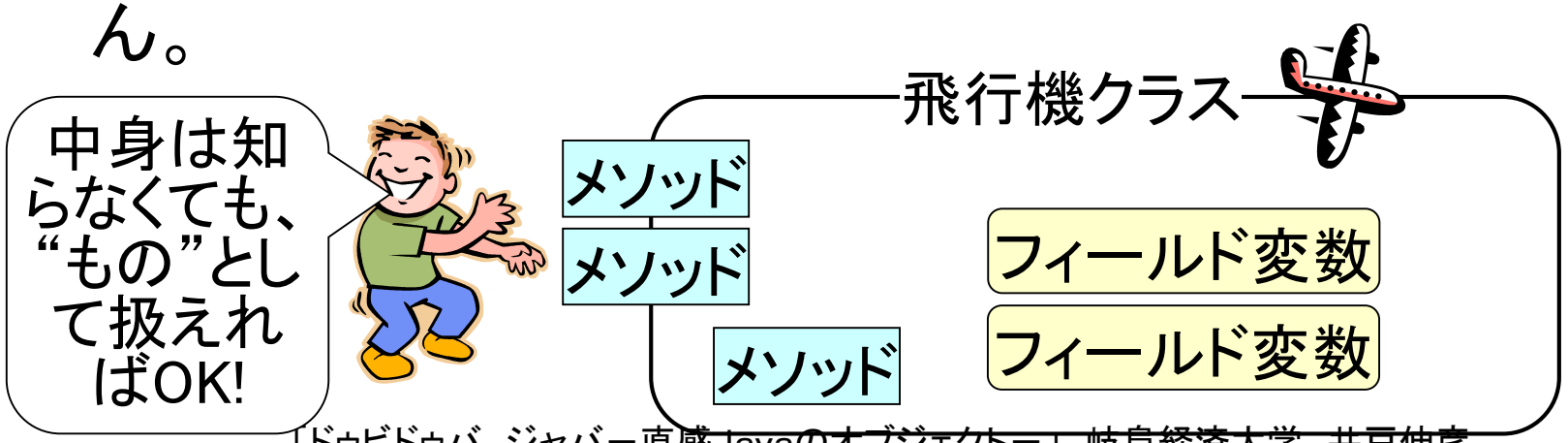
■しかしながら、Javaのようなオブジェクト指向言語は、次の例のような考えに基づき設計されています（このスライドでは詳細には触れません）。



（3．2．2）カプセル化

■クラスを“もの”と考えたとき、クラスの外に何を見せるかにより、メンバ（フィールド変数とメソッド）を“**public**”と“**private**”とのいずれにするかを決めることになります。

■カプセル化の考えでは、「“もの”の内部構造は隠して、“もの”として扱う上での単純な操作を外に見せる」ようにします。詳細は省略しますが、この考えから一般的にはフィールド変数を“**public**”とすることはありません。



（3．3．1）パッケージとアクセス

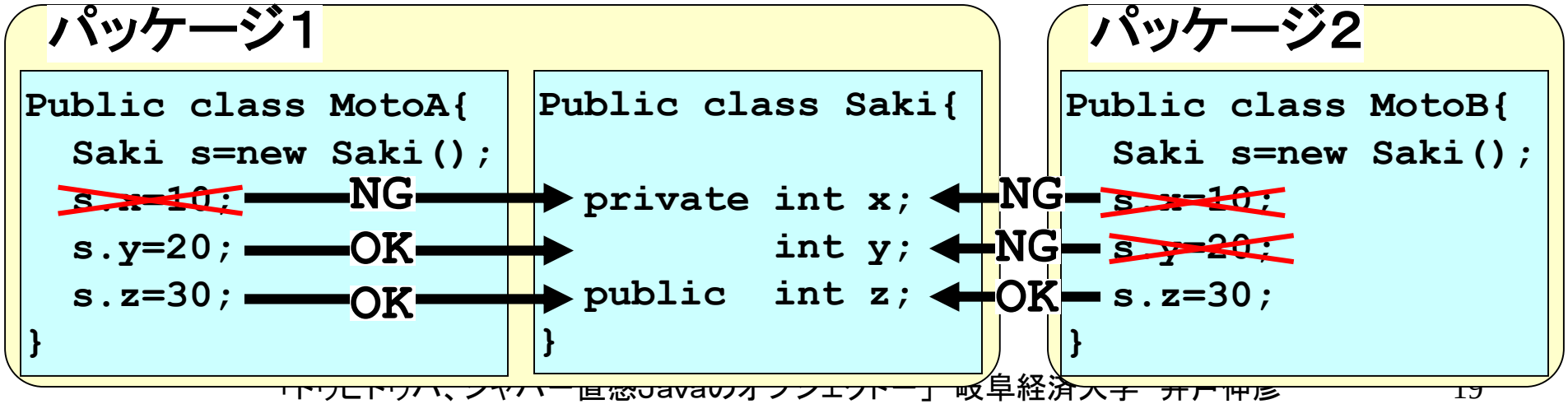
■アクセス先であるフィールド／メソッドへのアクセス修飾子については、次の3種類があります。

- **public / private /**（何も付けない）

■アクセス元については、次の区別があります（パッケージについてスライド(7)で勉強した際、再度確認してください）。

- クラスの内外の区別
- パッケージの内外の区別

■アクセスの可否をまとめると次のようになります。

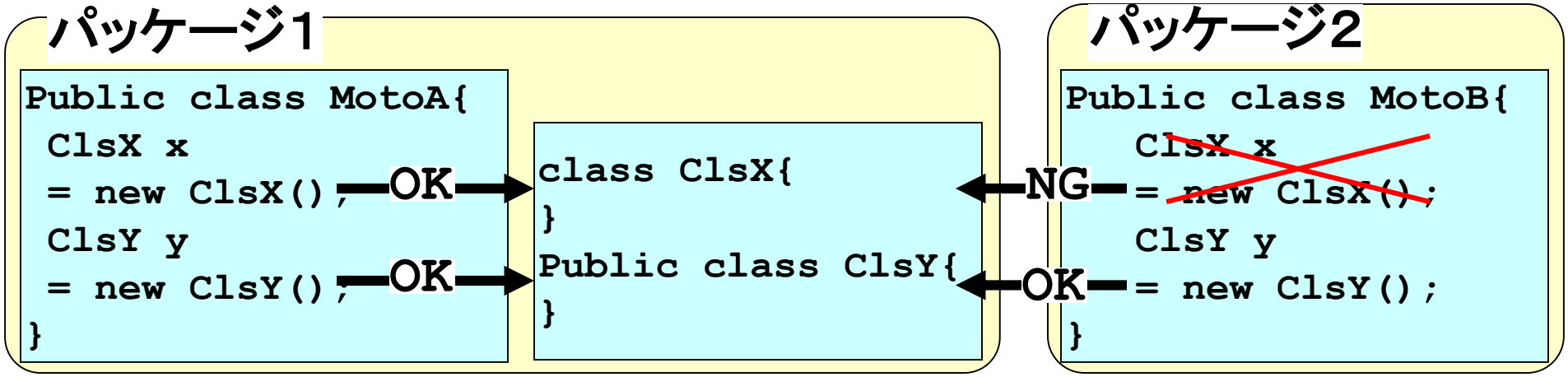


（３．３．２）クラス名につくアクセス修飾子

■クラスにも次の2種類のアクセス修飾子があります。
- **public / （何も付けない）**

■クラスへのアクセス元については、次の区別があります。
- パッケージの内外の区別

■アクセスの可否は、次のようになります。



■ひとつのファイルには、publicのクラスはひとつだけ許容されます。スライド(2)の２つのファイルは、ひとつのファイル（“FootballTeam.java”）にまとめてもOKです。

（3．4）パラメタとフィールド

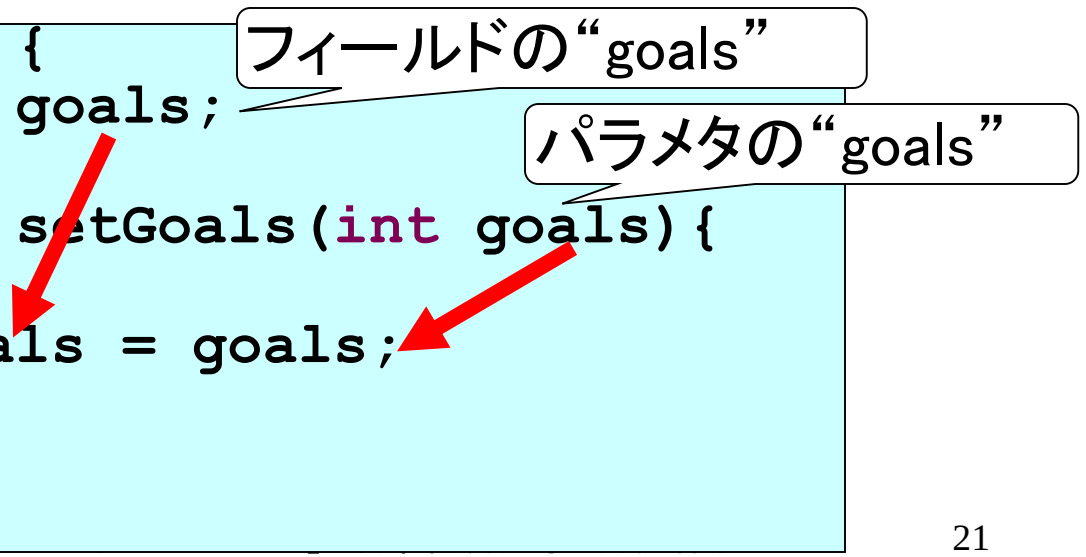
■クラス“Player2”のメソッド“setGoals”では、
`this.goals = goals;` という実行文があります。

■スライド(2.5)に記したとおり、`this.goals` はフィールドの“goals”を指します。

■同じ名前のパラメタとフィールドがある場合、パラメタのほうが優先されるので、“this.”が付されていないgoalsはパラメタを指しています。

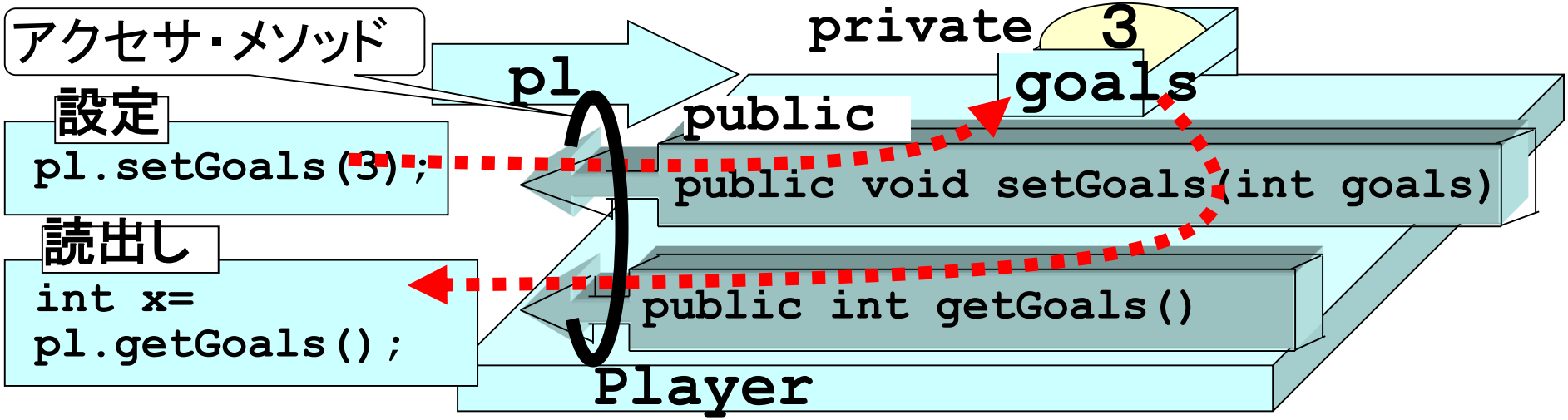
■このような記し方はよく使われるので、覚えておいてください。

```
class Player2 {
  private int goals;
        :
  public void setGoals(int goals){
        :
      this.goals = goals;
        :
  }
}
```



（３．５．１）アクセサ・メソッド

■フィールド変数の値を設定するメソッドと値を読み出すメソッドは、アクセサメソッドと呼ばれます。



■アクセサメソッドの名前は、次のような規則に従い、フィールド変数名から決めことになっています。

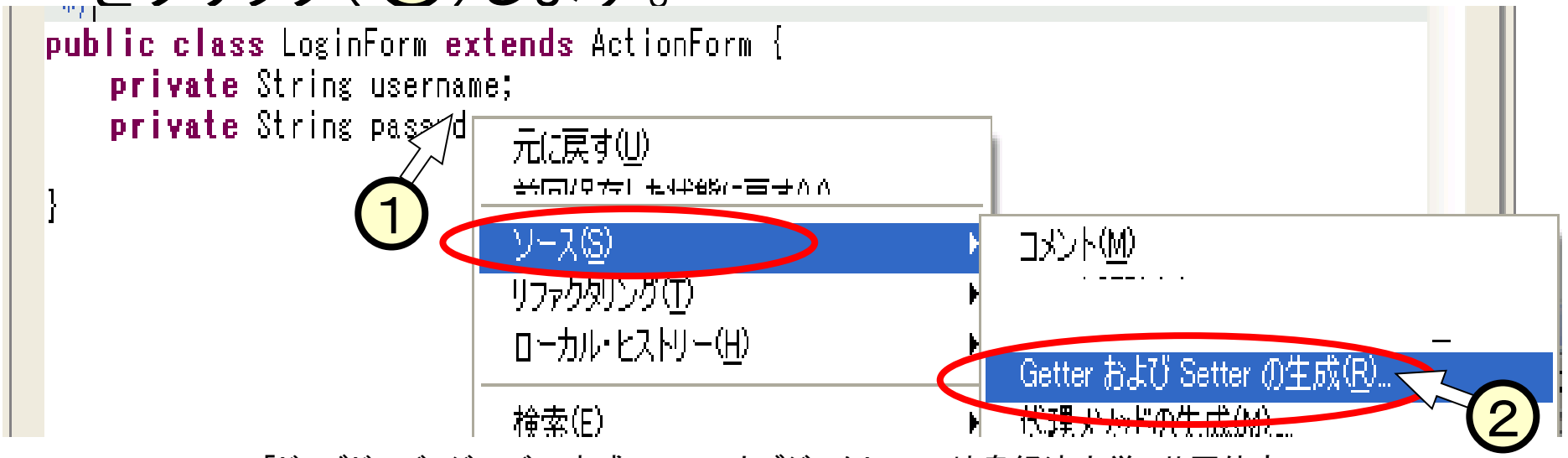
- 設定するメソッド（セッター：setter）
  ⇒　set+(最初の文字を大文字にしたフィールド名）
- 読み出すメソッド（ゲッター：getter）
  ⇒　get+(最初の文字を大文字にしたフィールド名）

（3.5.2）eclipseでのアクセサ・メソッド編集ー1ー

■アクセサメソッドの編集は、eclipse上で簡単に行うことが出来ます。

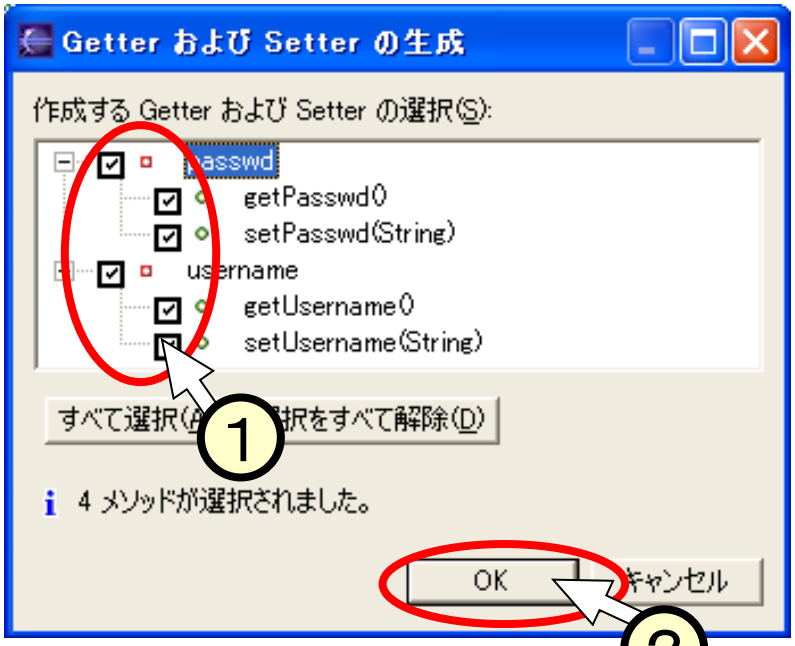
■フィールド変数（下図ではusernameとpasswd）の宣言のところまでは、入力したとします。

■エディター上のソースを右クリック（①）し、ポップアップメニューから、[ソース]-[GetterおよびSetterの生成]をクリック（②）します。



（3.5.3）eclipseでのアクセサ・メソッド編集―2―

- 「GetterおよびSetterの生成」のウインドウにて、作成するアクセサにチェック（①）します。
- [OK]をクリック（②）します。
- 変数“username”のアクセサ・メソッドが自動的に編集されます（③）。




```
 */
public class LoginForm extends ActionForm {
    private String username;
    private String passwd;

    /**
     * @return
     */
    public String getPasswd() {
        return passwd;
    }

    /**
     * @return
     */
    public String getUsername() {
        return username;
    }

    /**
     * @param string
     */
    public void setPasswd(String string) {
        passwd = string;
    }

    /**
     * @param string
     */
    public void setUsername(String string) {
        username = string;
    }
}
```

③

（３．６）課題

■課題３－１（課題２－１のカプセル化版の作成）

- スライド（3.1.1）のプログラムを作成してください。
- フィールド“name”も加えて、課題２－１のような出力を得るように変更してください。ただし、nameのフィールドも“**private**”とします。

■課題３－２（課題２－２のカプセル化版の作成）

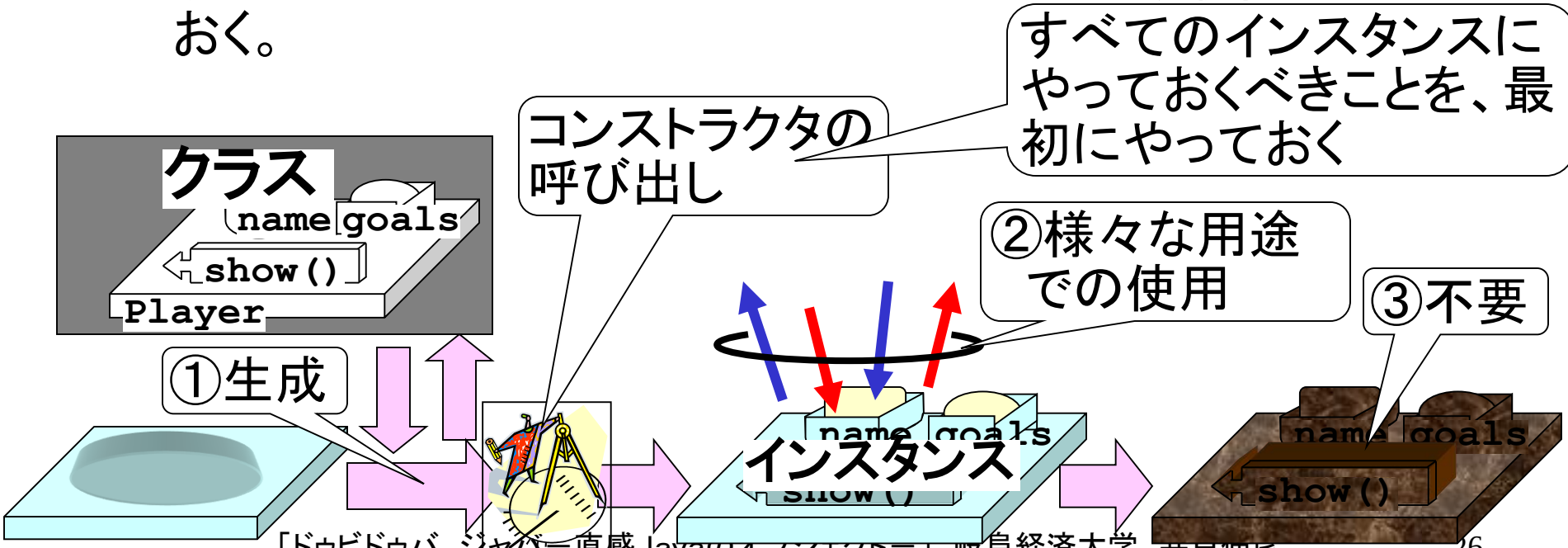
- 課題２－２のBaseballTeamクラス、BPlayerクラスを、次のように変更してください。
  - ◆フィールドは、“**private**”とする。
  - ◆守備位置(position)には、次の値以外は入らないようにする。
    - “投手”、“捕手”、“内野手”、“外野手”
    - 上記以外の値の場合は、“不明”と設定しておく。
  - ◆得点(run)には、負の値が入らないようにする。
- 得点をひとつ増やす次のようなメソッドをBPlayerクラスにつくり、BaseballTeamクラスから呼び出してみてください。
  - ◆**public void incrementRuns()**

（4）コンストラクタ

■インスタンスは、①生成されて、②様々や用途で使用され、③最後に不要となるという生涯を送ります。

■コンストラクタとは、インスタンスが生成される際に呼び出される特別なメソッドです。次のような処理を行います。

- すべてのインスタンスにやっておくべきことを、最初にやっておく。



（４．１．１）コンストラクタの例（１／２）

■コンストラクタは、次の特徴があります。
- クラス名と同じ名前である。
- 戻り値がない（voidも不要）。
- オブジェクトが生成されたときに自動的に呼び出される。

■パラメタの異なる2つ以上のコンストラクタを作ることもあります。

■次の例（次スライドに続く）を見ていきます。

<FootballTeam3.java>

```
public class FootballTeam3 {
  public static void main(String[] args){
    Player3 playerA,playerB;
    playerA = new Player3();
    playerB = new Player3("高原",3);
    playerA.show();
    playerB.show();
  }
}
```

（4．1．2)コンストラクタの例(2／2)

<Player3.java>

```
public class Player3 {
  private String name;
  private int goals;
  Player3(){
    name="[noname]";
    goals=0;
  }
  Player3(String name,int goals){
    this.name=name;
    setGoals(goals);
  }
  public void show(){
    System.out.println("名前："+name);
    System.out.println("得点："+goals);
  }
  public void setGoals(int goals){
    if(goals<0) this.goals = 0;
    else        this.goals = goals;
  }
  public void setName(String name){
    this.name=name;
  }
}
```

クラス名と同じ名前のメソッド⇒コンストラクタ

同上

<実行結果>

```
$ java FootballTeam3
名前：[noname]
得点：0
名前：高原
得点：3
```

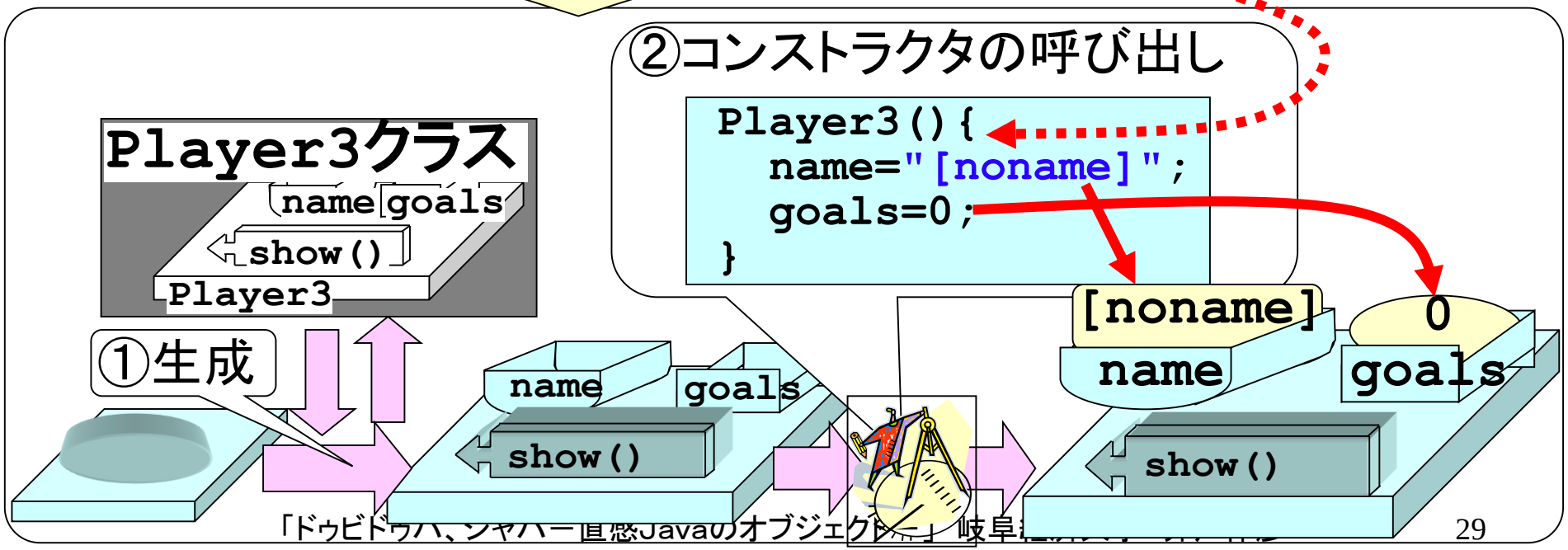
（4．2．1）ひとつめのインスタンスの初期化

■スライド(4.1)のプログラムのコンストラクタでは、フィールド変数の初期化を行っています。

■ひとつめのインスタンス（“playerA”）は、引数のないコンストラクタにより、次のように初期化されます。

```
playerA = new Player3();
```

パラメタの無いコンストラクタ



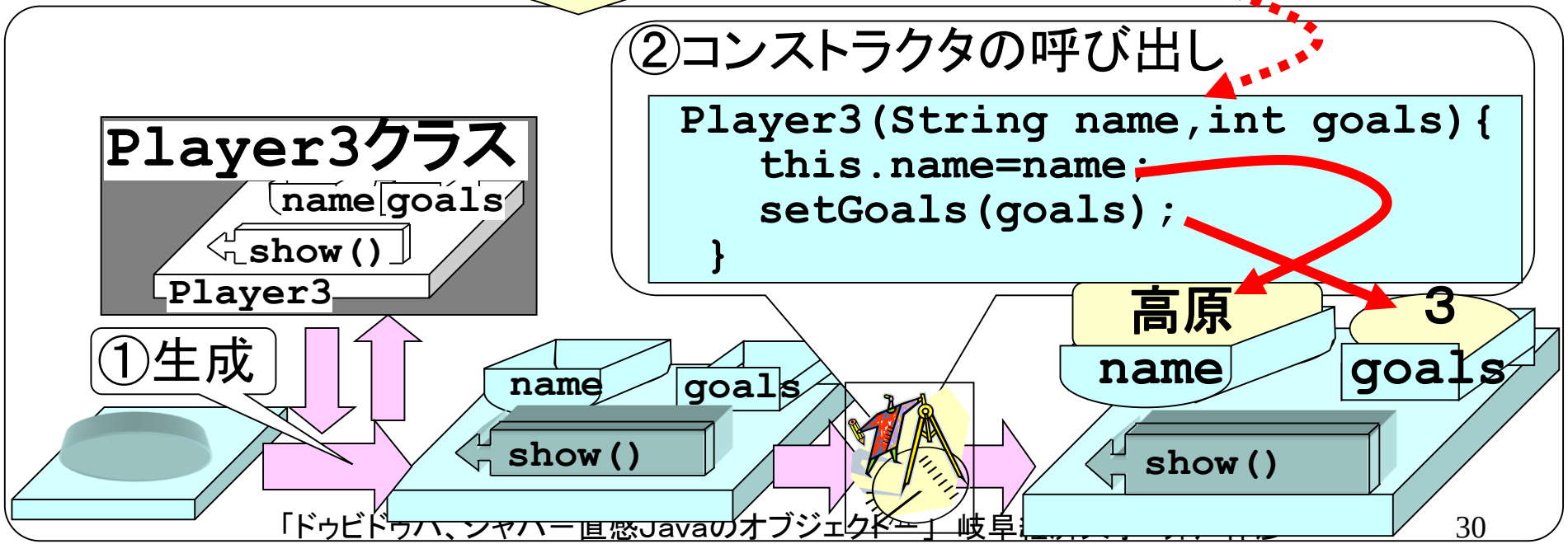
（4．2．2）ふたつめのインスタンスの初期化

■ふたつめのインスタンス（“playerB”）は、2つの引数を持ったコンストラクタにより、初期化されます。

（一般にJavaでは、同じ名前で異なるインスタンスを持つメソッドは、別のものとして扱われます。）

```
playerB = new Player3(“高原”,3);
```

パラメタ2つのコンストラクタ



（４．３．１）課題

■課題４－１

- スライド(4.1.1)(4.1.2)のプログラムを作成してください。
- 次のようなコンストラクタをPlayerクラスに作成してください。
  - ◆パラメタは1つ。該パラメタにより、名前を設定する。
  - ◆得点には“0”を設定する。
- FootballTeamクラスから上記のコンストラクタを利用してみてください。

（４．３．２）課題

■課題４－２

- 課題3-2のBPlayerクラスに次のコンストラクタを追加し、BaseballTeamクラスで該コンストラクタを利用してみてください。
  - ◆コンストラクタ1
    - ロパラメタはなし。
    - ロ名前/守備位置/得点にそれぞれ“unknown”/”unknown”/”0”の初期値を与える。
  - ◆コンストラクタ2
    - ロパラメタ2つ：名前、守備位置
    - ロ名前/守備位置にパラメタの初期値を与え、得点には“0”の初期値を与える。

（５．１）既に使っていたクラスメソッド

■ここまでその意味については特に説明せず、次のようなプログラムを使ってきました。

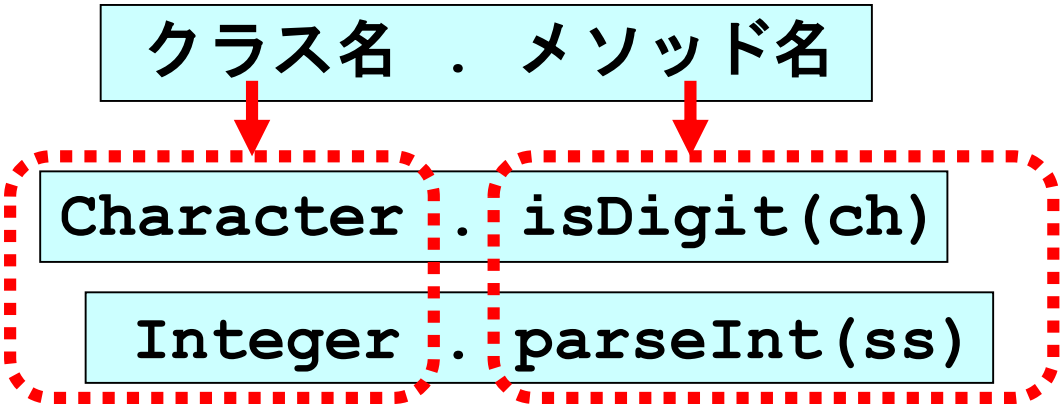
「覚書スライド(13.2)」

```
if(Character.isDigit(ch)){   //chが数字ならば真
```

「覚書スライド(13.3.1)」

```
d = Integer.parseInt(ss);
```

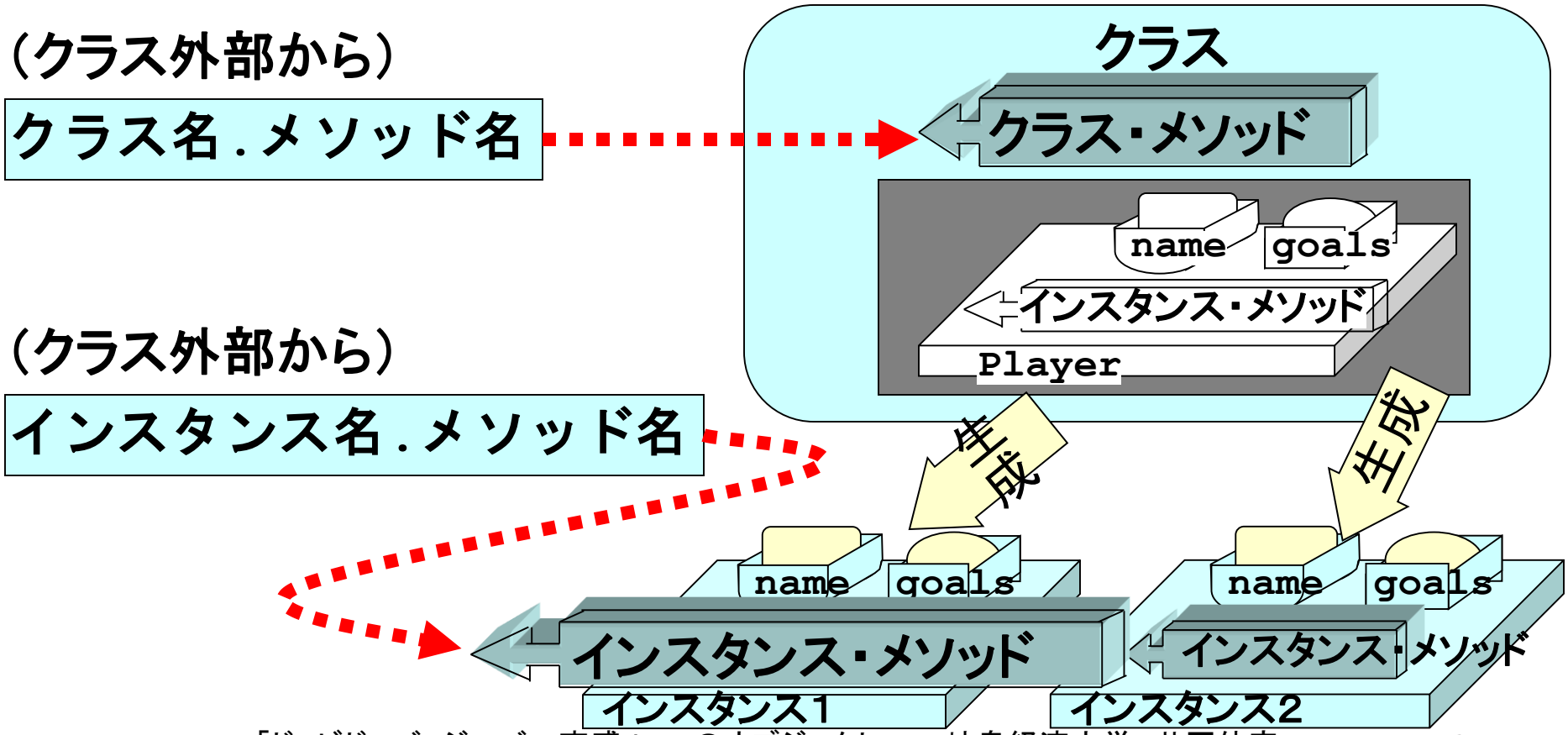
■実は、これらはクラス・メソッドと呼ばれるもので、クラス外部からは次のような形で利用します。



（５．２）クラス・メソッド

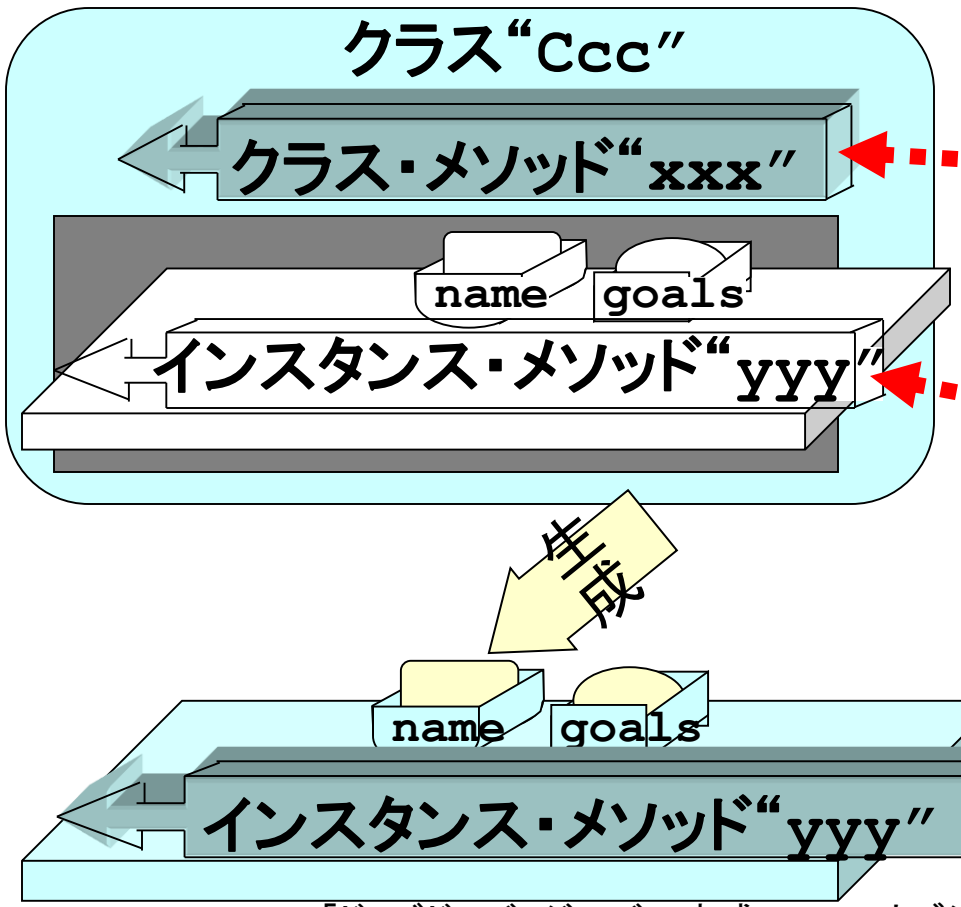
■クラス・メソッドは、インスタンスに関係なく呼び出すことが出来るメソッドです。

■クラス・メソッドと区別する際、いままで見てきた普通のメソッドを、インスタンス・メソッドと呼びます。



（５．３）static修飾子

■クラス・メソッドは、static修飾子を付けることで、インスタンス・メソッドと区別します。



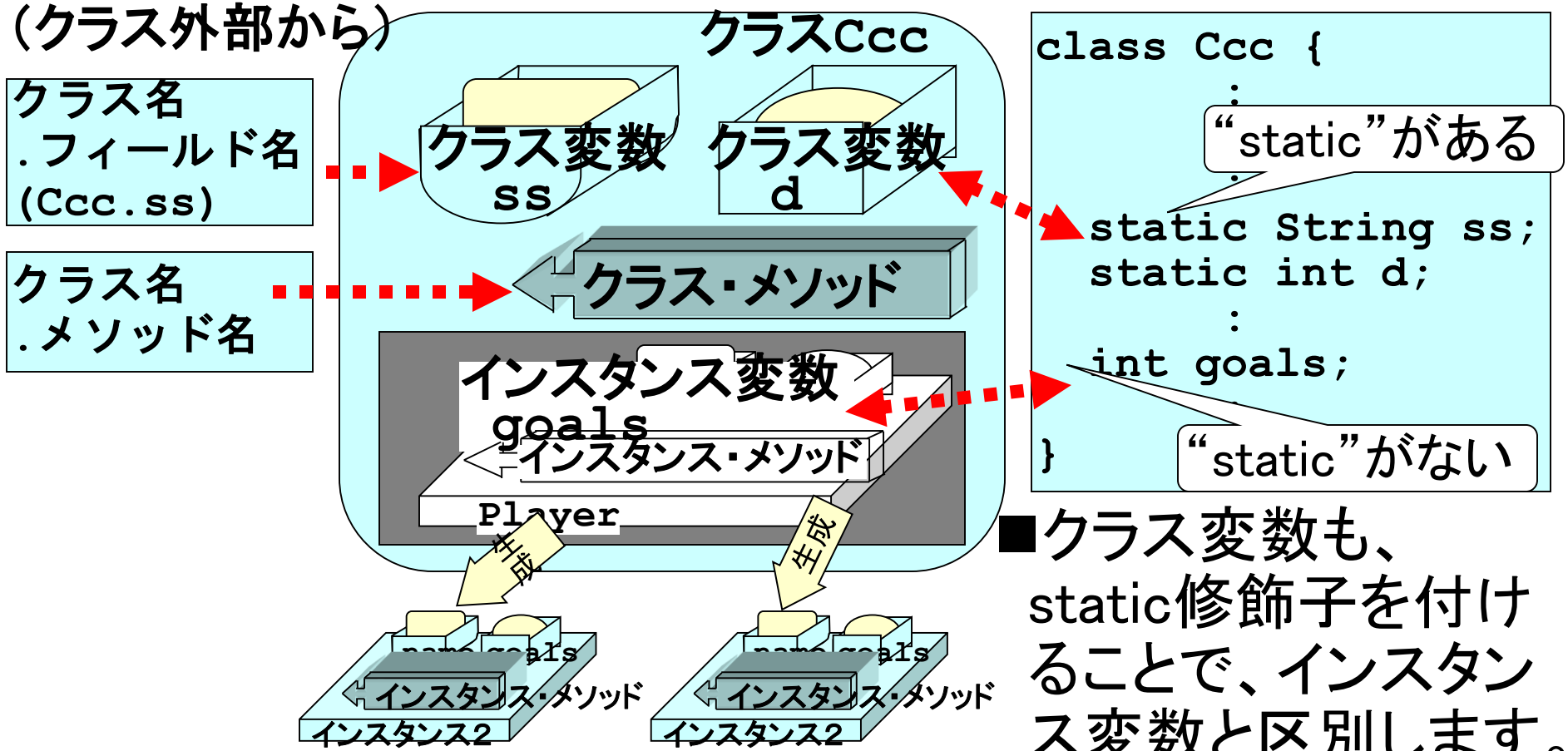

```
class Ccc {
        :
        :
  static void xxx(){
    //クラス・メソッド
  }
        :
        :
  void yyy (){
    //インスタンス・
    // メソッド
  }
}
```

“static”がある

“static”がない

（5．4）クラス変数

■クラス・メソッドがあるように、インスタンスとは関係なくクラスに1つだけ存在する変数を定義することが出来ます。これを、クラス変数と言います。



■クラス変数も、static修飾子を付けることで、インスタンス変数と区別します。

（5．5）小さな整理

■static修飾子の有無により、メソッドと変数は次のように呼ばれます。

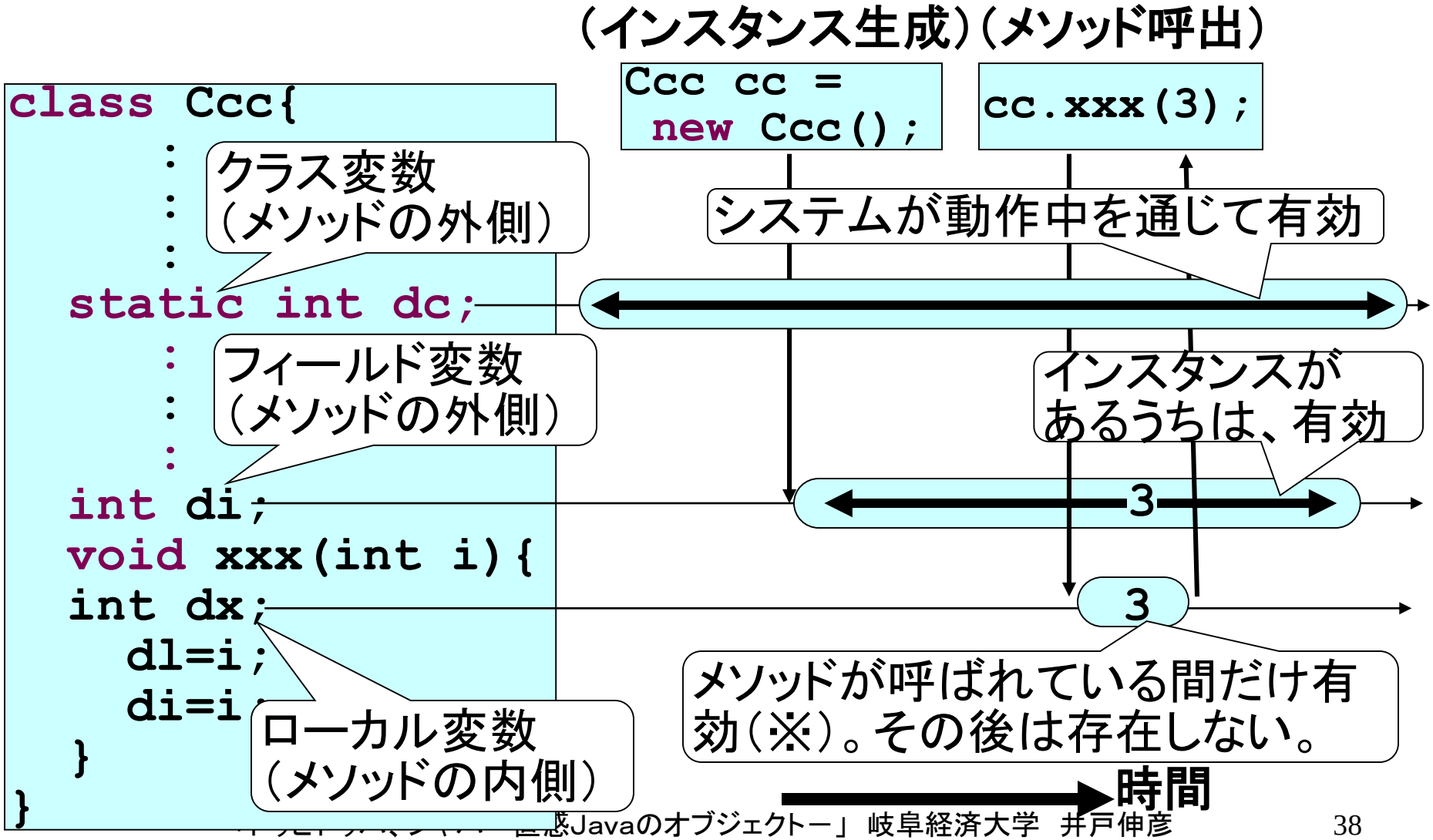
<table>
<tr><td rowspan="4">メンバ
（広義の
クラス
・メンバ）</td><td rowspan="2">インスタンス
・メンバ
（static無し）</td><td>インスタンス
メソッド</td><td></td></tr>
<tr><td>インスタンス
変数</td><td rowspan="2">フィールド
変数</td></tr>
<tr><td rowspan="2">クラス
・メンバ
（static有り）</td><td>クラス変数</td></tr>
<tr><td>クラス
メソッド</td><td></td></tr>
</table>

■**public/private/（指定なし）**の区別は、staticの有無と独立に存在します。すなわち、次のいずれも用います。

- “**public static int x;**”、“**private static int x;**”、“**static int x;**”、
- “**public int x;**”、“**private int x;**”、“**int x;**”

（５．６）クラス変数の寿命

■クラス変数は、システムの動作中を通じて、有効です。



(5.7)クラス・メソッド／変数の例

<Ex5.java>

```
public class Ex5 {
  public static void main(String[] args) {
    System.out.println("円周率は"+Circle.PI+"です。");
    double r = 2.0;
    System.out.print("半径"+r+"の円の面積は、");
    System.out.println(Circle.area(r)+"です。");
  }
}
```

<Circle.java>

```
public class Circle {
  public static final double PI = 3.14;
  public static double area(double r){
    return r*r*PI;
  }
}
```

<実行結果>

```
$ java Ex5
円周率は3.14です。
半径2.0の円の面積は、12.56です。
```

（5．8）final定数

■前スライドのCircleクラスのクラス変数“PI”は、finalと修飾されています。

```
public static final double PI = 3.14;
```

- “final”の付いた変数は、値が変更出来きません。すなわち、定数ということになります。

■例えば、次の例のように、プログラム中に直接数値を書き込むことは良くありません。

```
if(x>31){
```

■“31”が何を表す数なのかを解りやすくするため、定数を用いて書くようにします。

大文字が普通

ここを変更すれば、プログラム中のすべての値が変更できる。

クラス変数でなくてもfinalは使用出来る。

```
final int MAX_DAY=31;
        :
if(x>MAX_DAY){
```

日付の最大値であることが分かる。

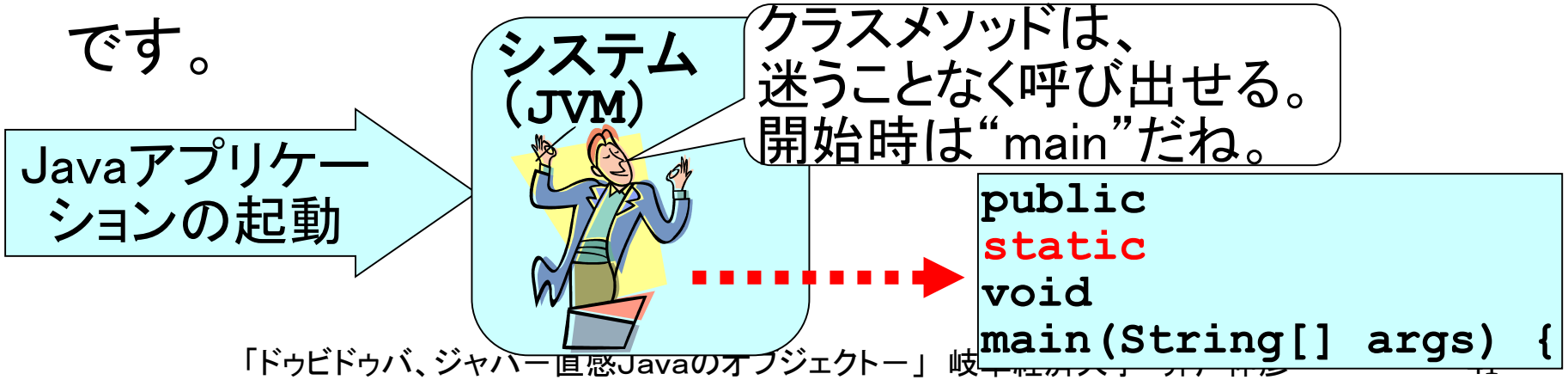
（5．9）どんな場合にクラスメンバを使うのか？

■クラスメンバ（クラスメソッドとクラス変数）はどのような場合に使ったら良いのでしょうか？

■実は既にクラスメソッドは繰り返し使っています。

```
public class FootballTeam {
  public static void main(String[] args) {
      :
```

■**main**メソッドは、クラスメソッドです。すなわち、インスタンスとは関係なく呼び出すことが出来、システムは最初にこのメソッドを呼び出す約束になっているという訳です。

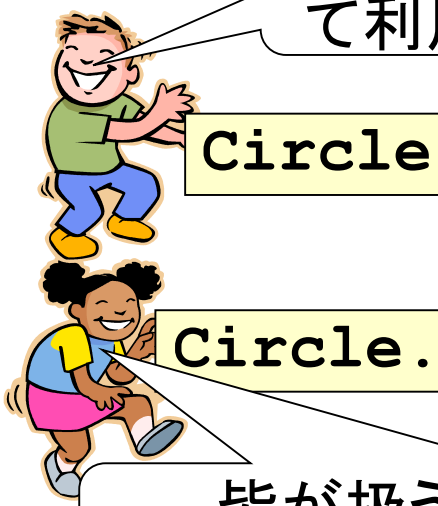


（5．9．1）どこからでも迷わず呼び出せる

■mainがそうであったように、どこからでも迷わず呼び出したいような性質を持ったメンバ（変数/メソッド）を、クラス・メンバとすれば良いわけです。

■このような性質を持ったものとしてまず考えられるのは、スライド（5.7）に示した例のような、次の場合です。

（実際、円周率は、“**Math.PI**”として提供されています。）



共通の定数として利用しよう！

Circle.PI

Circle.area(r)

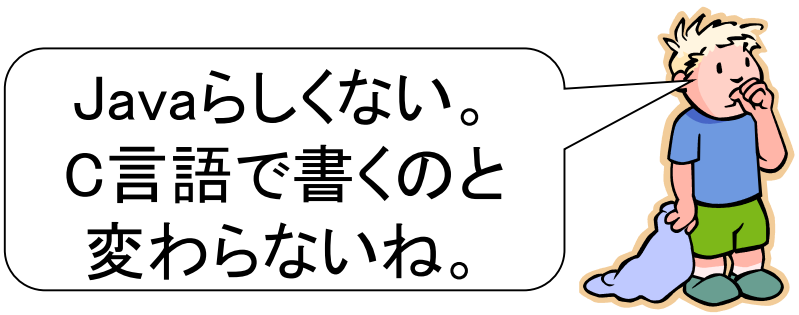
皆が扱う便利な道具として、クラス・メソッドを利用しよう！

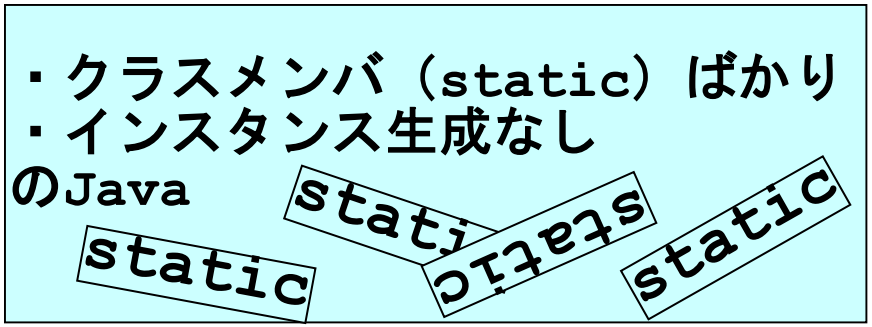
```
public class Circle {
  public static final double PI = 3.14;
  public static double area(double r){
    return r*r*PI;
  }
}
```

（５．９．２）staticばかりのJavaプログラム？

■次のような極端なJavaのプログラムを考えて見ます。
- すべてのメンバをクラスメンバとして、インスタンスの生成などは行わない。

■上記のように制限したプログラムでも、あらゆる機能を実現することは可能です。しかしながら、そのプログラムは、Javaらしい（オブジェクト指向らしい）プログラムではなく、C言語のようなプログラムになります。





■スライド(5.9.1)のような場合以外は、「クラス・メンバは使用しないほうが良い」と筆者は思います。

（５．１０）課題

■課題５－１

- スライド(5.7)のプログラムを作成してください。

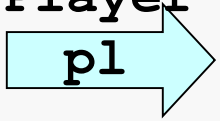
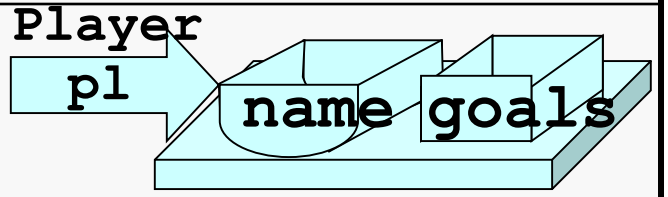
■課題５－２

- スライド(5.7)のプログラムに、次の値を求めるクラス・メソッドを追加し、Ex5クラスから使ってみてください。
  - ◆半径から円周（circumference）を求めるクラスメソッド。

(6)クラス型の変数

■既に見たように、生成したインスタンスは、クラス型の変数に保持していました。

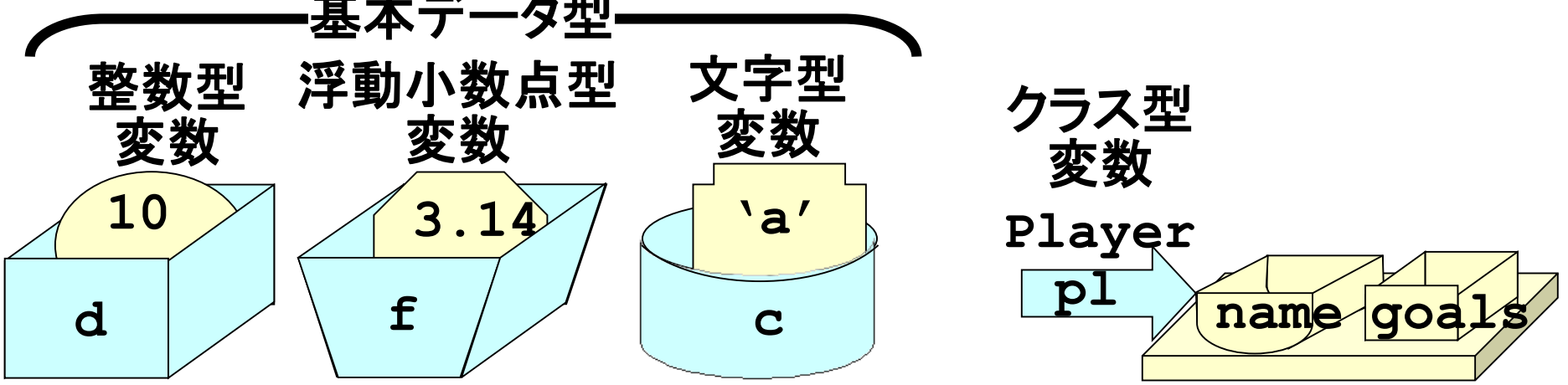
- 下記の例では、**Player**クラス型の変数“**p1**”に、生成したインスタンスを保持しています。

操作	記述	操作後の状態
インスタンスへのインデックス(クラス型の変数)を用意する	`Player p1;`	
インスタンスを生成し、インデックスに関連付ける	`p1=new Player();`	

■クラス型の変数は、**int,char,double** などの基本データ型の変数と同じように扱うことが出来ます。

（６．１）基本データ型との比較

■基本データ型がそれぞれの性質を持つデータを保持するもの（箱）であったように、クラス型変数は定義されたクラスの性質を持つインスタンスを保持します。




```
int d;
d = 10;
```

```
double f;
f = 3.14;
```

```
char c;
c = 'a';
```

```
Player pl;
pl = new Player();
```

小数という性質のデータを保持する。

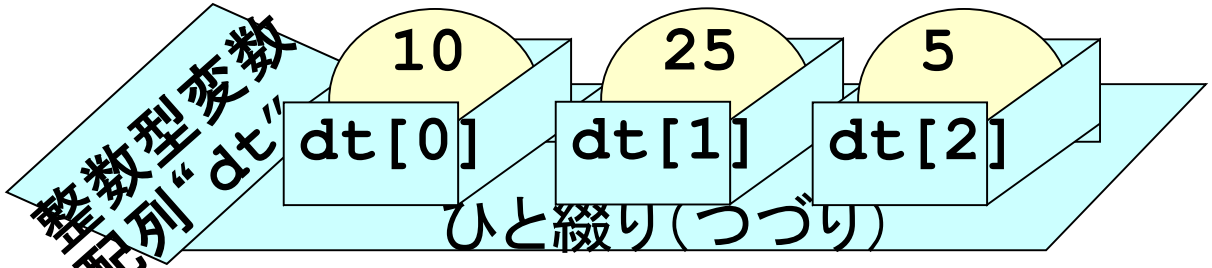
類似

Playerクラスで定義された性質のインスタンスを保持する

（6．2）クラス型変数の配列

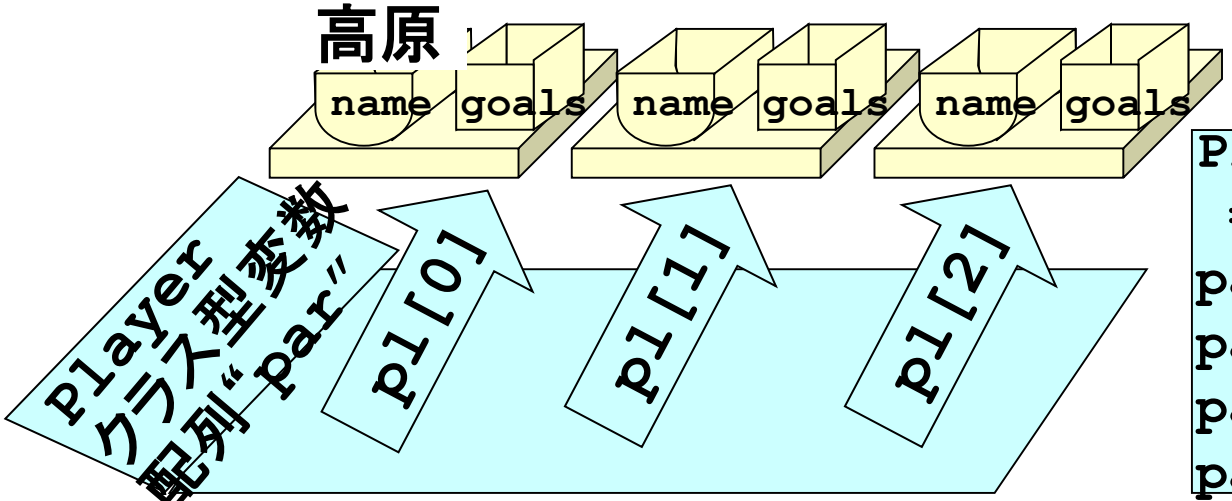
■基本データ型と同様に、クラス型変数の配列も定義することができます。

＜整数型変数の配列＞




```
int dt[]
 = new int[3];
dt[0] = 10;
dt[1] = 25;
dt[2] = 5;
```

＜(Player)クラス型変数の配列＞




```
Player par[]
 = new Player[3];
par[0] = new Player();
par[1] = new Player();
par[2] = new Player();
par[0].name=“高原”
```

（6．3．1）クラス型変数の引数

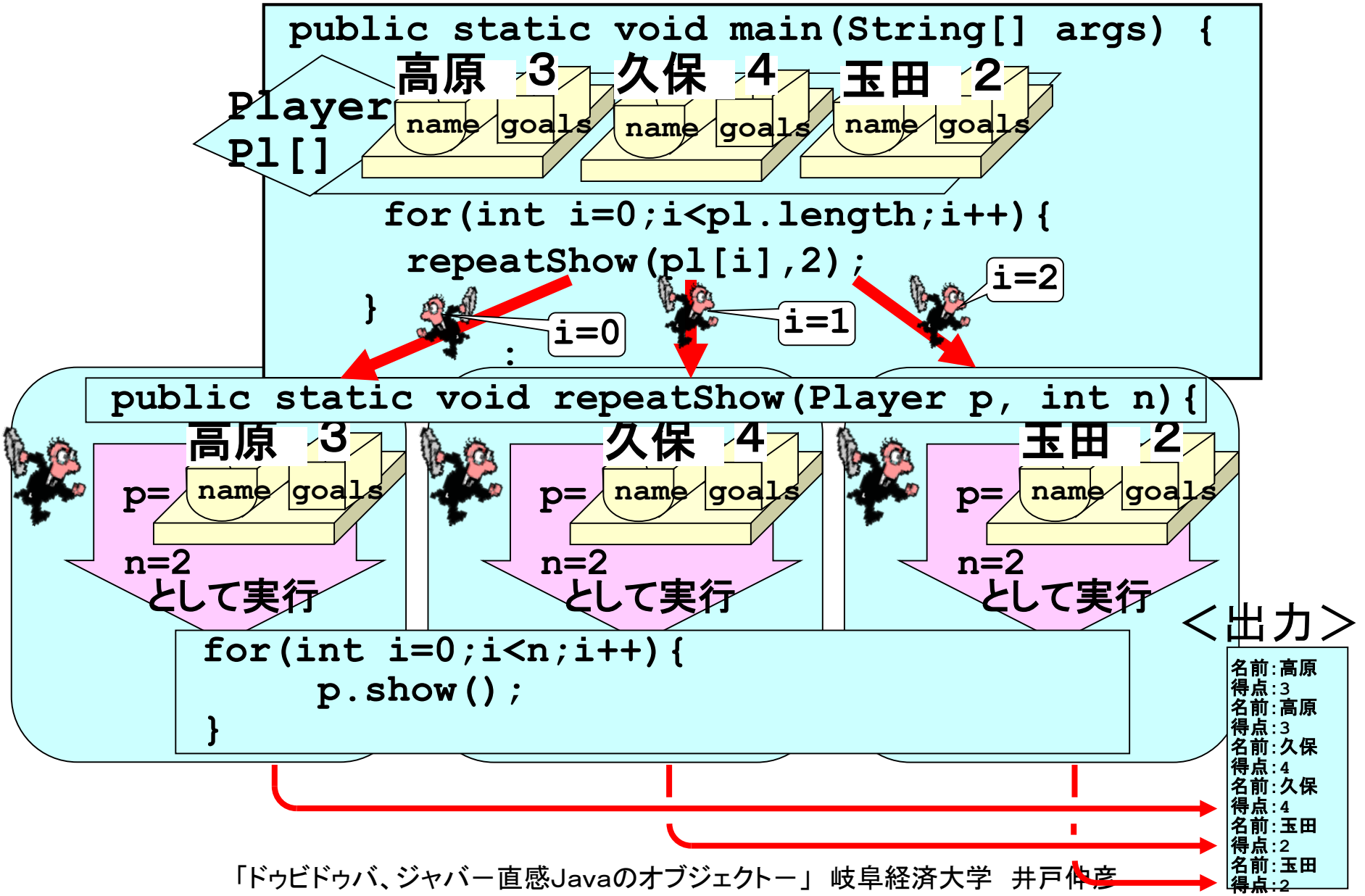
■メソッドの引数としても、クラス型変数を用いることが出来ます（戻り値として用いることも出来ます）。

```
public class FootballTeam6 {

  public static void main(String[] args) {
    Player3 pl[] = new Player3[3];
    pl[0] = new Player3("高原",3);
    pl[1] = new Player3("久保",4);
    pl[2] = new Player3("玉田",2);
    for(int i=0;i<pl.length;i++){
      repeatShow(pl[i],2);
    }
  }

  static void repeatShow(Player3 p,int n){
    for(int i=0;i<n;i++){
      p.show();
    }
  }
}
```

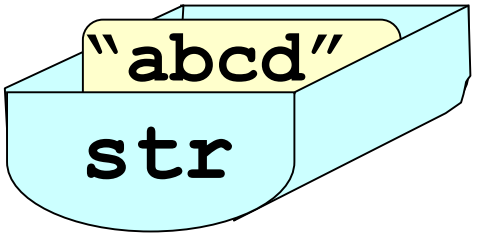
＜出力＞

```
名前:高原
得点:3
名前:高原
得点:3
名前:久保
得点:4
名前:久保
得点:4
名前:玉田
得点:2
名前:玉田
得点:2
```

(6．3．2)プログラムの動き



（6．4）Stringクラス



■ここまで文字列を扱う際に利用してきた、Stringはクラスです。

```
String str;
str = “abcd”;
```

■Stringクラスは標準クラスといい、Javaの環境であれば、どこでも利用できます。

■スライド「Java覚書」(13)で学んだStringの用法は、予め用意されたStringクラスの機能だったということです。

```
String ss;
```

```
if(ss.equals(“abc”)){
char c = str.charAt(2);
int n = ss.indexOf(“ab”);
```

■Stringクラスでは、インスタンス生成を明示的に書く必要は無かった訳です（書いてもOKです）。

```
String ss = “abc”;
```

同じ

```
String ss = new String(“abc”);
```

（６．５．１）課題

■課題６－１

- スライド(6.3.1)のプログラムを作成してください。

■課題６－２

- 次の配列を使って、スライド(6.3.1)のインスタンス生成の部分を、for文の中で行うようにしてください。

```
String nameAr[]={"高原","久保","玉田"};
int goalAr[]={3,4,2};
```

■課題６－３

- スライド(6.3.1)のプログラムを、次のように改造します。
  - ◆もっとも得点（goals）の多い選手のみを１回出力する（同じ得点の場合はどちらの選手でもOKとします）。
- Player3クラスには、次のメソッドを追加します。

```
public int getGoals(){
  return this.goals;
}
```

（次スライドへ続く）

（6．5．2）課題

■課題6－3（前スライドから続き）

- //????の部分を埋めて、プログラムを完成させてください。

```
public class BestForward {
  public static void main(String[] args) {
    Player3 pl[] = new Player3[3];
    pl[0] = new Player3("高原",3);
    pl[1] = new Player3("久保",4);
    pl[2] = new Player3("玉田",2);
    Player3 bestFwd = pl[0];
    for(int i=1;i<pl.length;i++){
      // ????  ここでメソッド"betterFwd"を用いる
    }
    System.out.println("良いフォワードは。。。");
    // ????
  }
  static Player3 betterFwd(Player3 p1,Player3 p2){
    // ????
  }
}
```

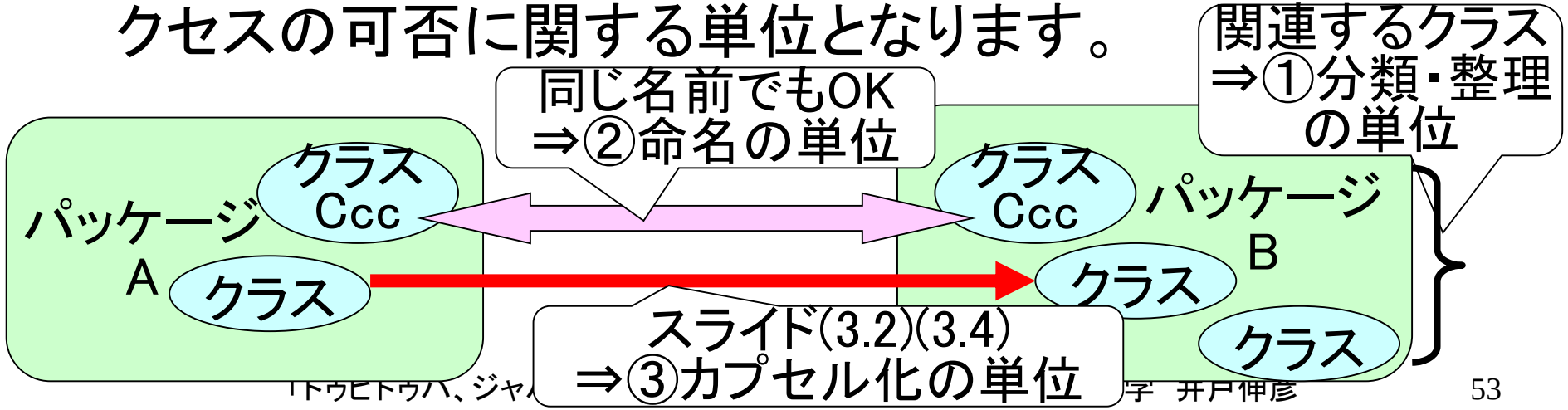
<実行結果>

```
$ java BestForward
良いフォワードは。。。
名前：久保
得点：4
```

Player3クラス変数を戻り値とするメソッド
（得点の大きい方を返す）

（7）パッケージ

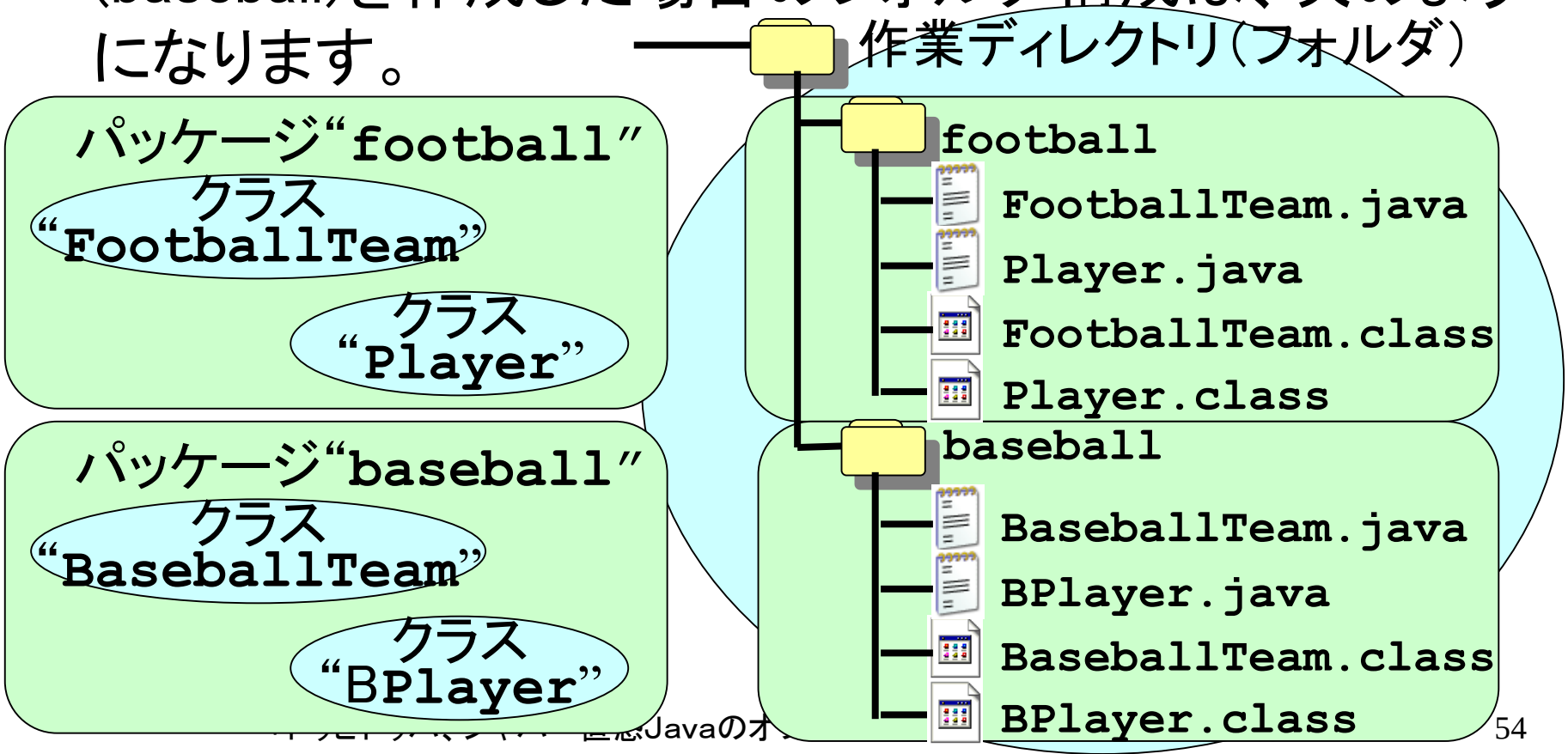
- さまざまなクラスを作成し、その数が多くなっていくと、クラスを分類して整理しておく手段が必要となります。
- ①パッケージは、クラスを分類・整理しておく手段です。関連するクラスを同じパッケージに入れておけば、扱いが便利ということになります。
- ②パッケージが異なれば、同じ名前のクラスがあっても問題は生じません。すなわち、クラスの命名の単位（＝名前空間）の役割も果たします。
- ③パッケージは、既にスライド(3.3)で示したように、アクセスの可否に関する単位となります。



（7．1．1）フォルダ（ディレクトリ）への格納

■同じパッケージのソースファイル（.java）とクラスファイル（.class）は、同じフォルダ（ディレクトリ）に格納します。

■フットボール・パッケージ(football)と野球パッケージ(baseball)を作成した場合のフォルダ構成は、次のようになります。



（7．1．2）ソースファイルでのパッケージ

■例えばパッケージ“**football**”に含まれるソースファイルには、ファイル先頭に次のようなパッケージを示す文を書きます。

<FootballTeam.java>

```
package football;

public class FootballTeam {
(略)
}
```

■なお、Javaではパッケージ名は小文字で始まる文字列とする習慣になっています。

（7．1．3）コンパイルと実行

■コマンドラインでのコンパイルと実行は、パッケージのフォルダを収容した作業ディレクトリ（フォルダ）から、次のように行います。

コンパイル時はファイルをパス名で指定

```
% cd 作業ディレクトリ
% javac football/FootballTeam.java
% java football.FootballTeam
```

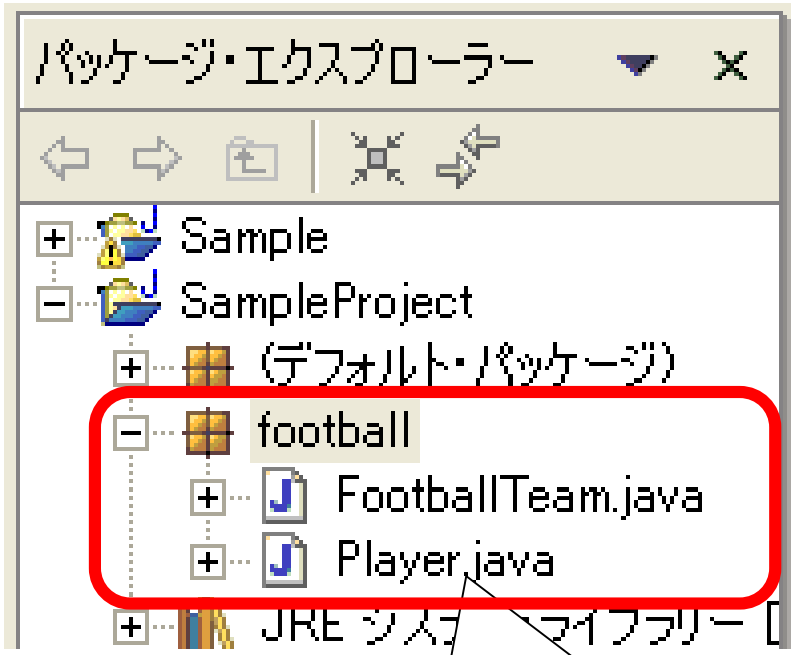
実行時は（パッケージ名．クラス名）で指定

（7．2．1）eclipseでのパッケージ操作

■eclipseでもパッケージ対応にフォルダが作成されることは同じです。

■次の操作について、説明します。

- パッケージの作成
（スライド(7.2.2)）
- パッケージ内のクラス作成
（スライド(7.2.3)）
- パッケージ内のクラスの実行
（スライド(7.2.4)）

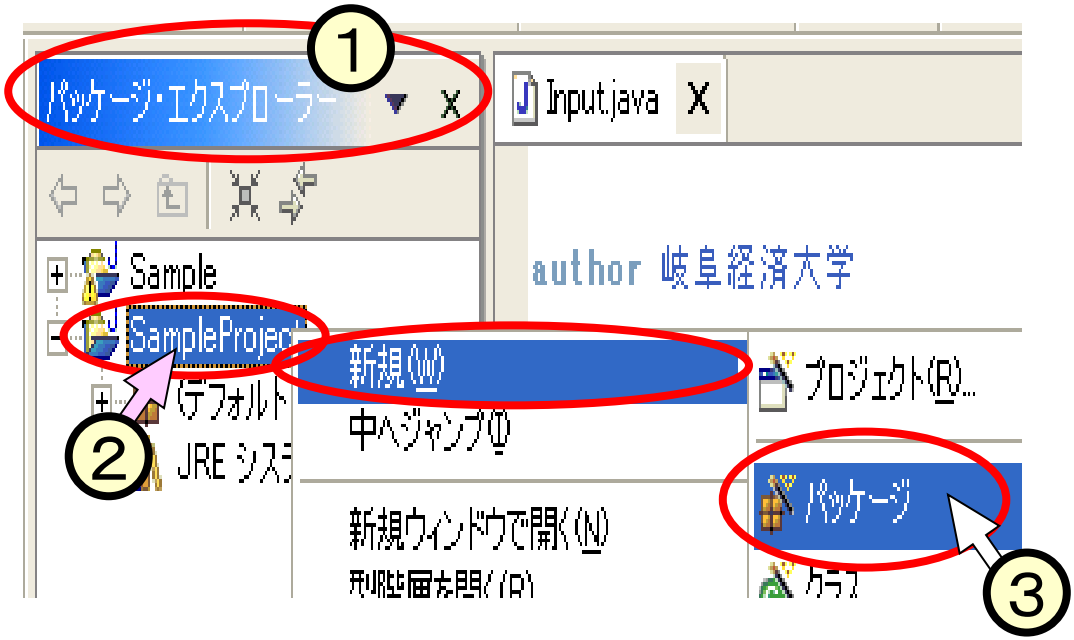


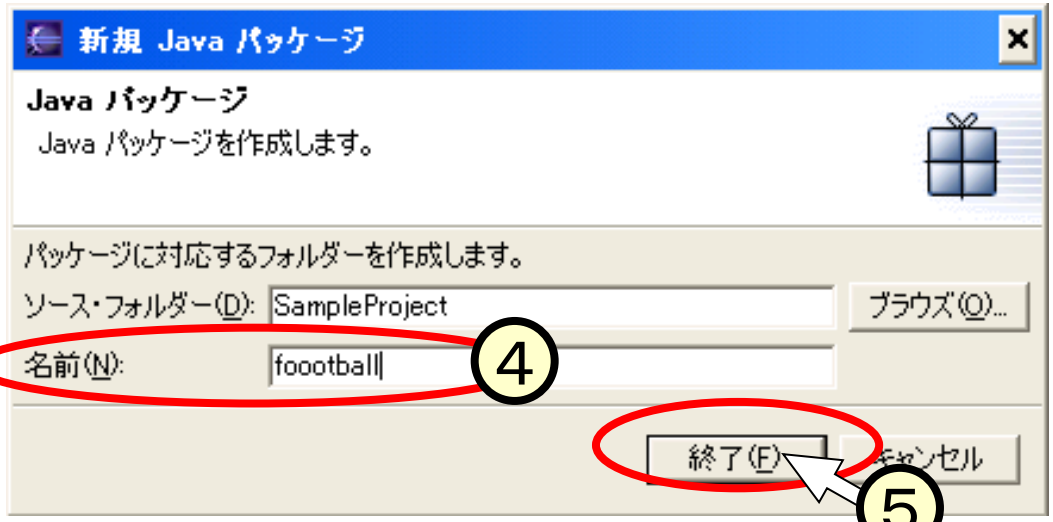
（7．2．2）eclipseでのパッケージ作成

■パッケージ・エクスプローラ(①)にて、パッケージを作成するフォルダを右クリック(②)し、ポップアップメニューから[新規]-[パッケージ]をクリック(③)します。

■「新規Javaパッケージ」ウインドウにて、名前欄にパッケージ名を入力(④)し、[終了]をクリック(⑤)します。

■前スライドのように、パッケージを示すフォルダが出来ます。

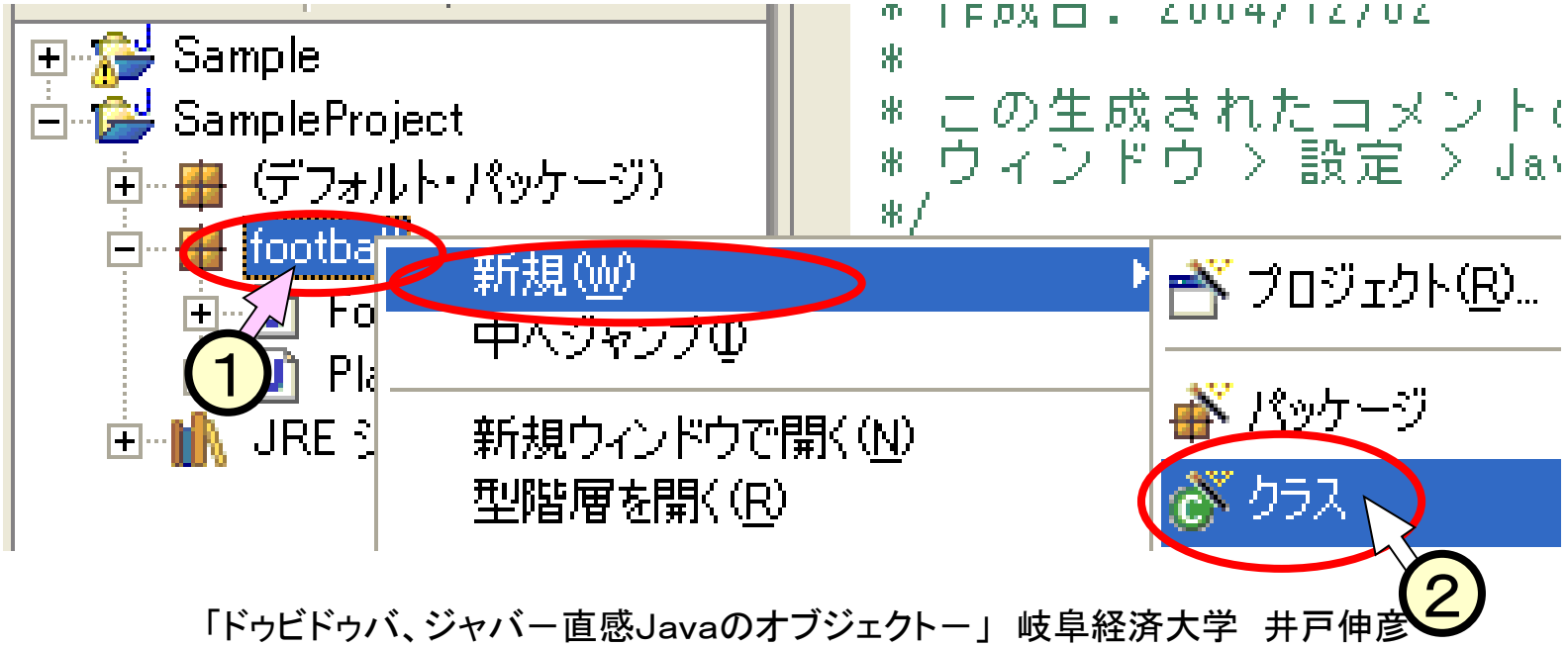




（7．2．3）パッケージ内のクラス作成

■パッケージ内にクラスを作成する際には、パッケージを右クリック（①）して、[新規]–[クラス]をクリック（②）します。後の操作は、今までパッケージを作成しなかった場合の操作と同じです。

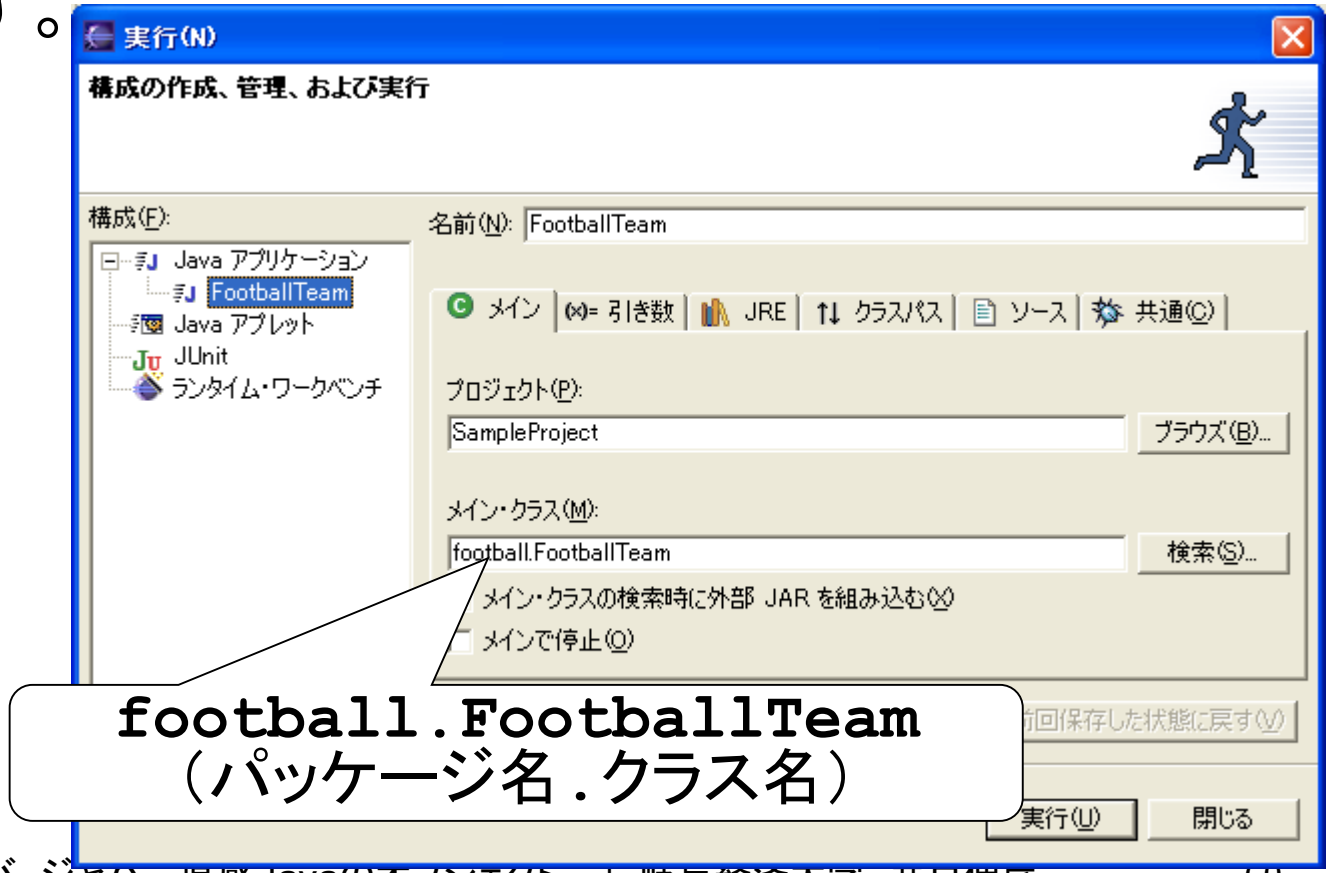
■このようにして作成したソースファイルには、“**package football;**”の行が自動で挿入されます。



（7．2．4）パッケージ内のクラスの実行

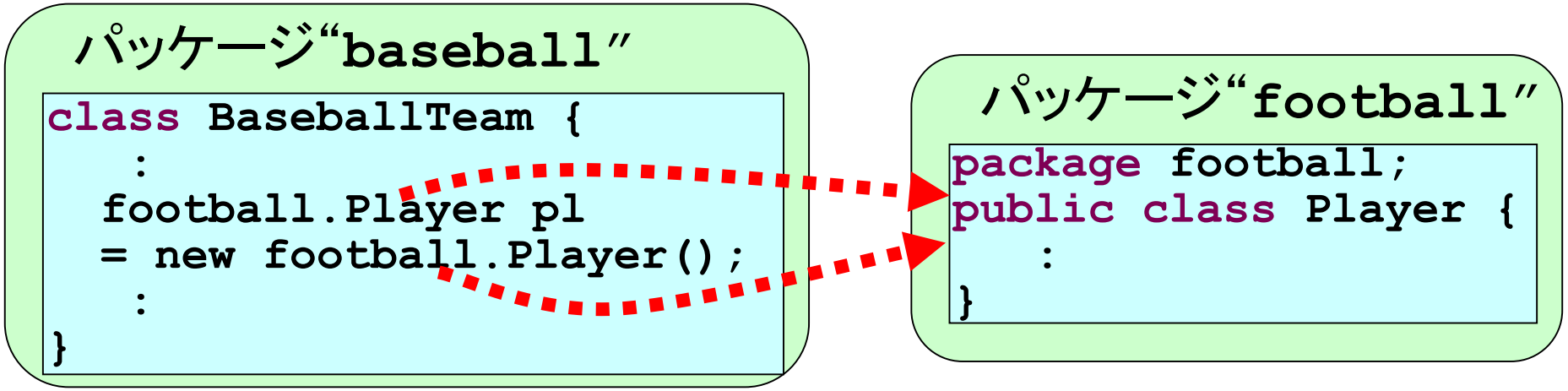
■実行のしかたも、パッケージを作成しなかった場合と同じです。

■実行するクラスの名前は、（パッケージ名.クラス名）で表示されます。



（7．3）他のパッケージのクラスの呼び出し

■プログラム中で他のパッケージのクラスを呼び出す場合、（パッケージ名.クラス名）でこれを指定します。



■スライド(3.2)(3.4)に記したように、パッケージを跨いだアクセスは、アクセス修飾子（**public/private/**なし）により、その可否が決まります。

■import(「Java覚書」のスライド(6.3)参照)により、パッケージ名を省略することが可能です。

（７．４）課題

■課題７－１

- ここまでに作ったクラスを、次のようにパッケージにまとめてください（前のクラスを消す必要はありません）。
  - ◆サッカーに関するクラス　⇒　footballパッケージ
  - ◆野球に関するクラス　⇒　baseballパッケージ

■課題７－２

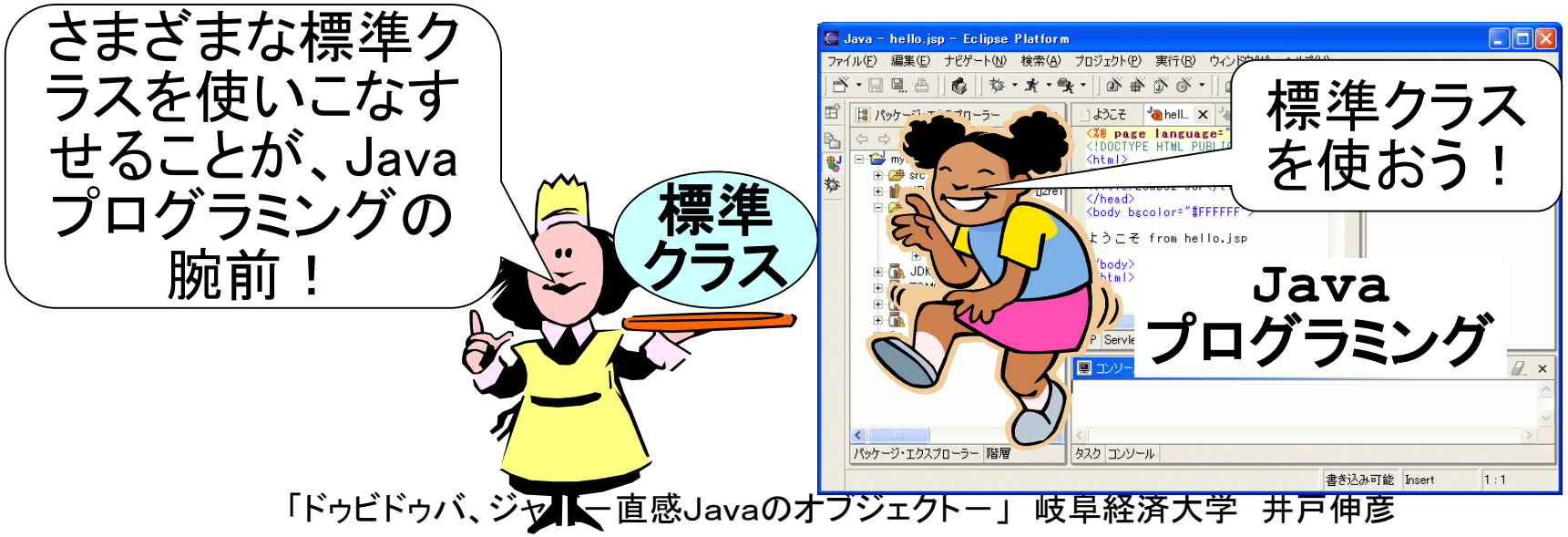
- baseballパッケージのBaseballTeamクラスを変更して、次のように野球とサッカーとの両方の表示を行うようにしてください。

```
$ java BaseballTeam
イチロー　外野手　１３０
松井　　外野手　１２０
野茂　　投手　　５
名前：高原
得点：３
名前：久保
得点：４
```

（8）標準クラス

■プログラムを作成する際に利用するクラスは、自分で作ったクラスに限る訳ではありません。他人が作ったクラスを大いに利用すべきです。

■Javaには標準クラスという一群のクラスが用意されていて、Javaが動く環境であればこれらのクラスを利用することができます。この標準クラスを利用することが、Javaでのプログラミングでは前提となっています。



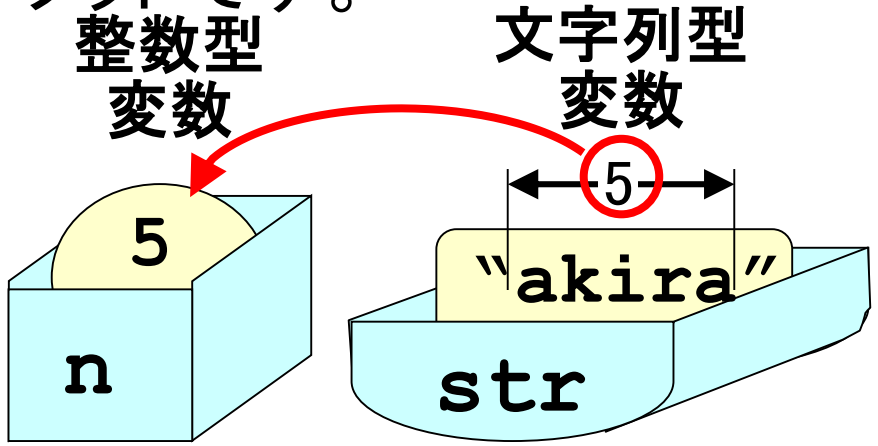
（8．1．1）既に使っている標準クラス　－1－

■ここまでも様々な標準クラスを使ってきています。

■スライド「Java覚書(13)標準のメソッド」で利用したメソッドは、実は標準クラスのメソッドです。

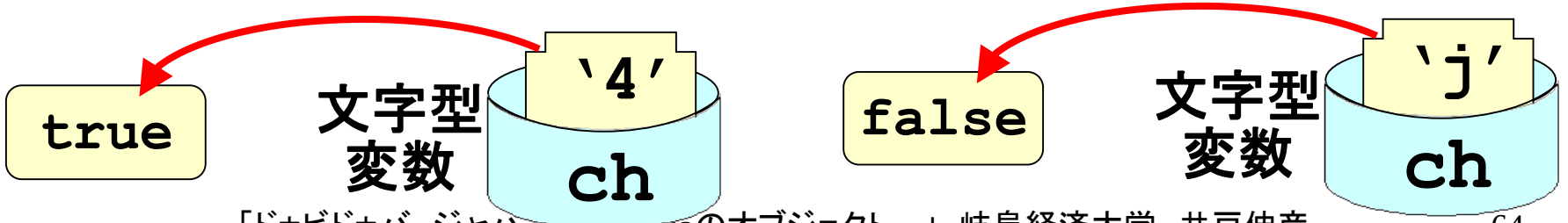
- 例えば。。。

```
String str = "akira";
int n = str.length();
// nには、数値の5が入る。
```



- 例えば。。。

```
if(Character.isDigit(ch)){   //chが数字ならば真
```



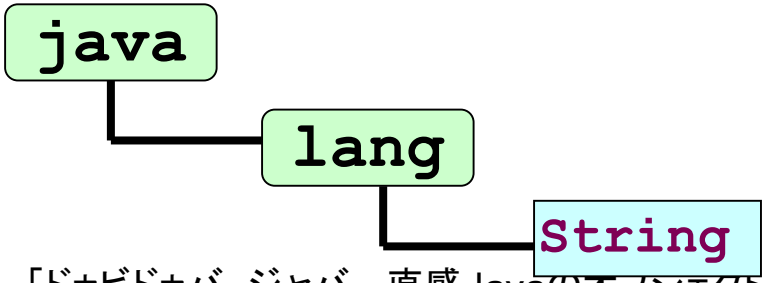
（8．1．2）java.langパッケージ

■Stringクラスなどの重要な標準クラスは、java.langパッケージに入っています。すなわち、本来ならば、Stringクラスは次のように記すはずです。

```
java.lang.String str = “akira”;
```

■しかしながら、java.langパッケージは基本的な機能をもち、使用頻度が高いので、すべてのJavaプログラムはデフォルトで次のインポートが宣言されていることになっています。 `java.lang.*;`

■java.langパッケージは、次のように階層化されています（多くのパッケージは階層化されています）。



（8．2．1）既に使っている標準クラス　－2－

■スライド「Java覚書(6)データの入力」では、いくつかの標準クラスを使いました。

```
BufferedReader kbd =
    new BufferedReader(new InputStreamReader(System.in));
```

kbd

キーボードから1行づつ読み取るのに、便利な“もの”、“kbd”を作った。

BufferedReader

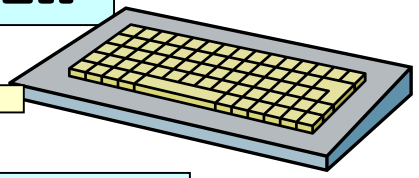
ひとまとめにする役割

InputStreamReader

文字コードを変換する役割

1文字づつ読み取る役割

System.in

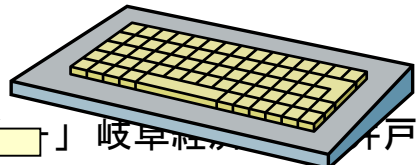


```
ss = kbd.readLine();
```

“入力した文字”
ss

“kbd”を使って、1行の文字列を読み出した。

kbd.readLine()

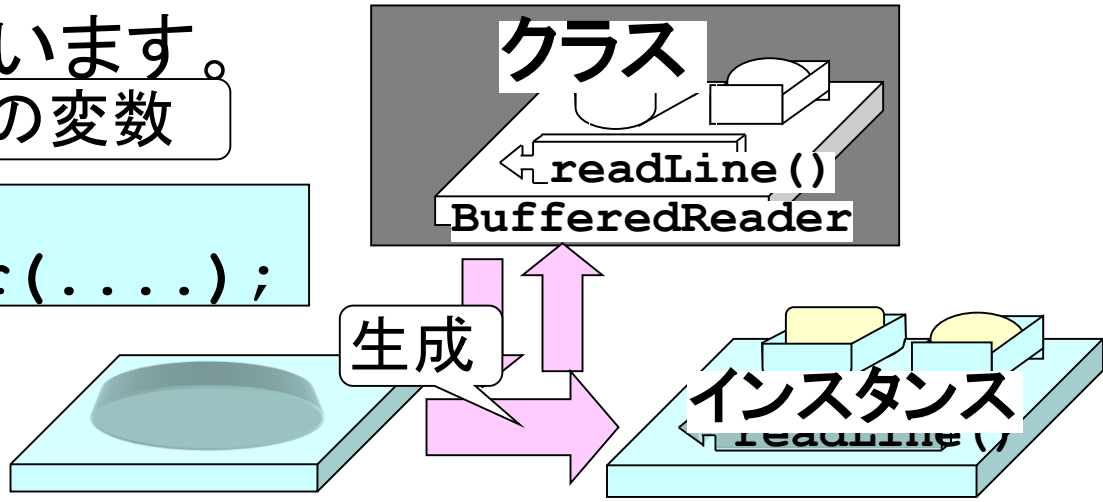


（8．2．2）インスタンス、コンストラクタ

■データの入力で利用していた“kbd”は、“BufferedReader”クラスのクラス型変数であったことが理解できると思います。

クラス型の変数

```
BufferedReader kbd
= new BufferedReader(....);
```



■“BufferedReader”クラスのインスタンス生成では、“InputStreamReader”クラスのインスタンスを引数とするコンストラクタが用いられています。

```
new BufferedReader(new InputStreamReader(System.in));
```

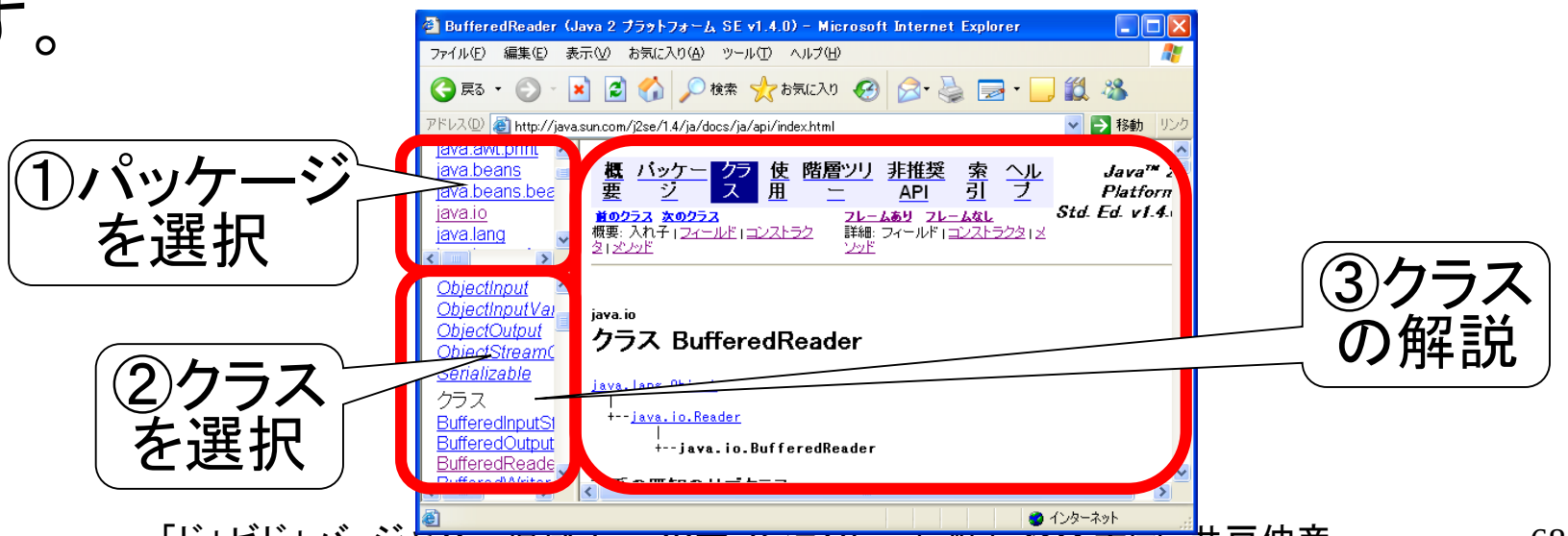
引数のあるコンストラクタ

(8．3)ドキュメント

■どのような標準クラスがあって、どのような使い方をするのかを説明したドキュメントは、オンライン上で参照出来ます(下記のサイト以外にも、多くのサイトに同一のドキュメントがアップされています。一般的には、その日本語ドキュメントをダウンロードすればOKです。)。

- http://java.sun.com/j2se/1.4/ja/docs/ja/api/index.html

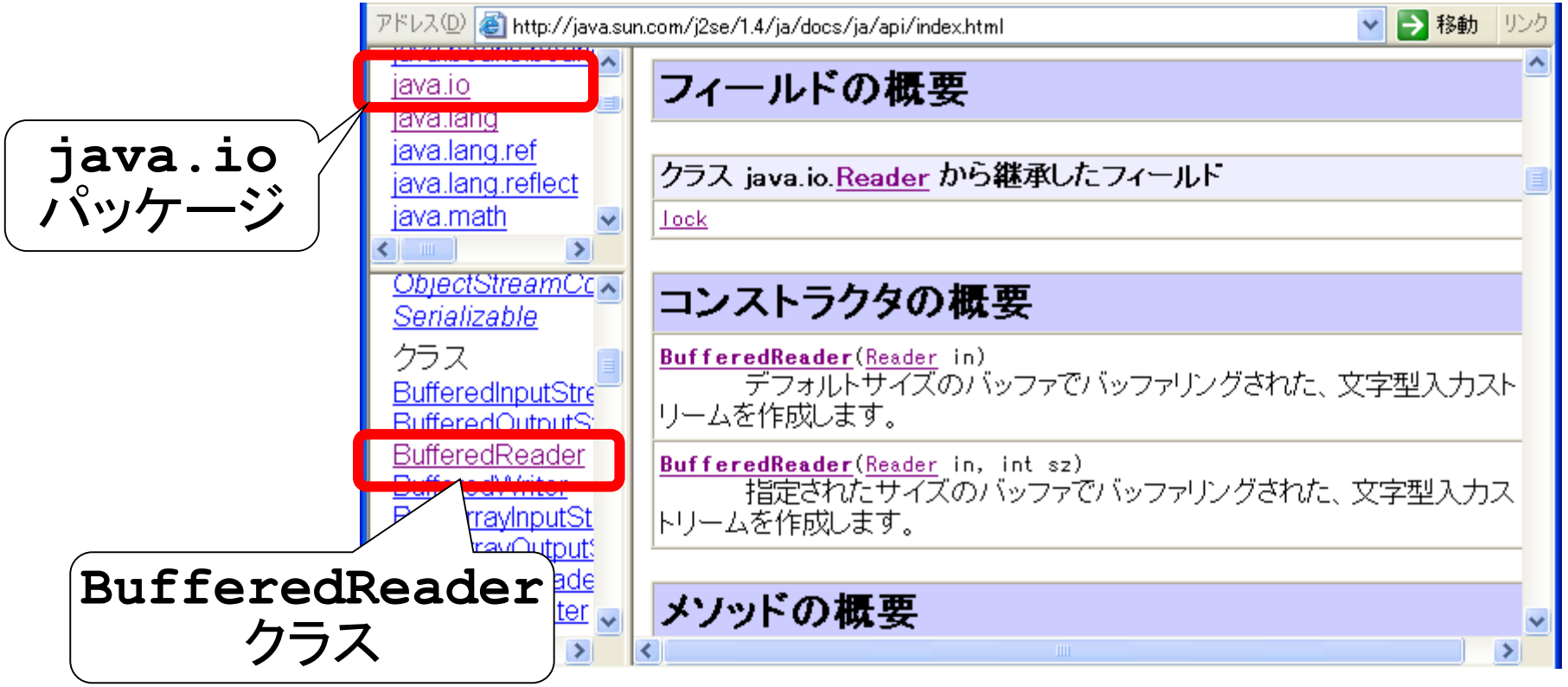
■ドキュメントでは、サブフレームでパッケージ(①)とクラス(②)とを選択することで、クラスの解説(③)を得ます。



（8．3．1）“BufferedReader”を見てみる

■BufferedReaderクラスをオンラインドキュメントで見てみます。

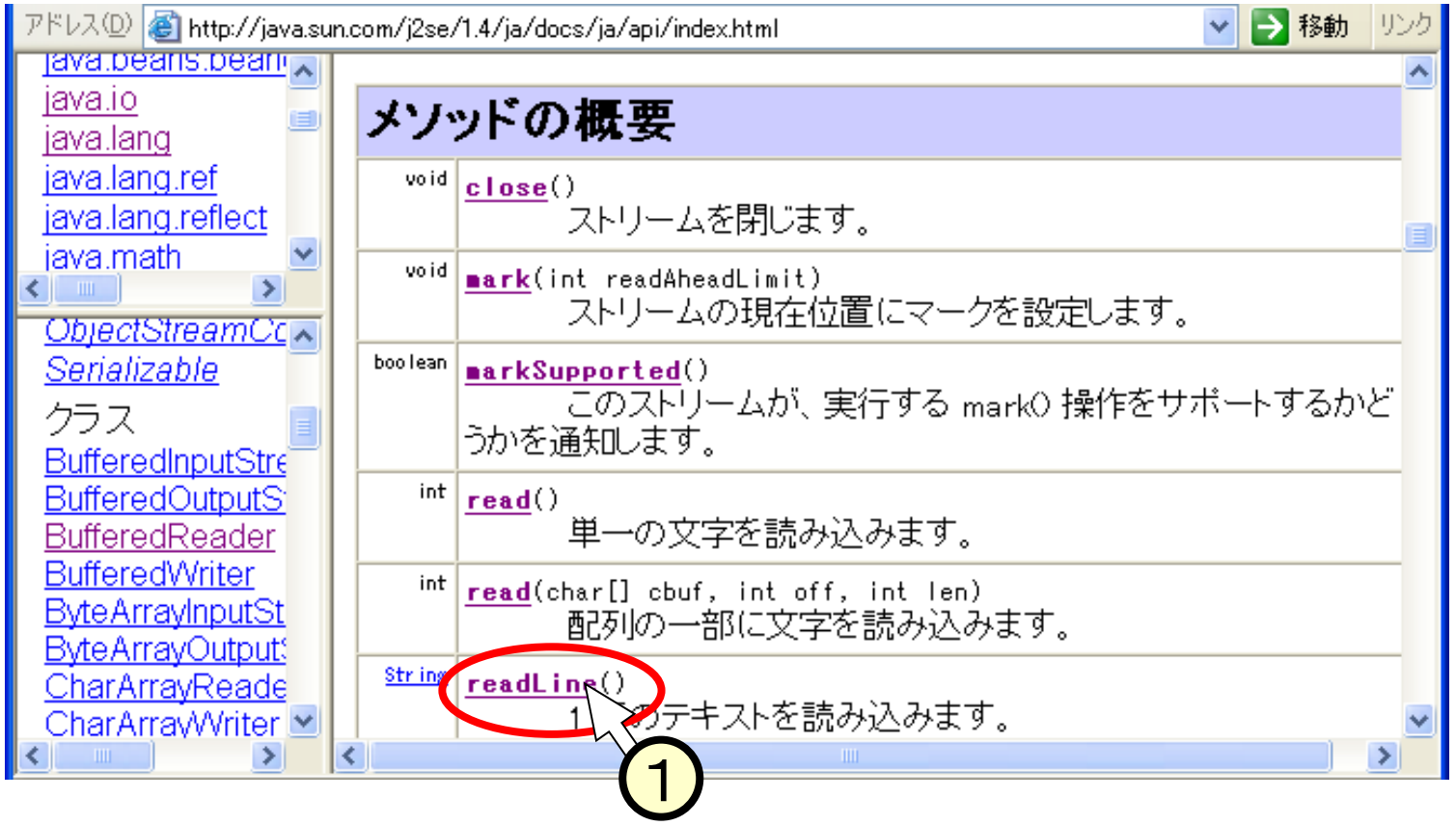
■クラスの解説の中には、「フィールドの概要」、「コンストラクタの概要」、「メソッドの概要」の一覧表があります。



（8．3．2）メソッドの概要

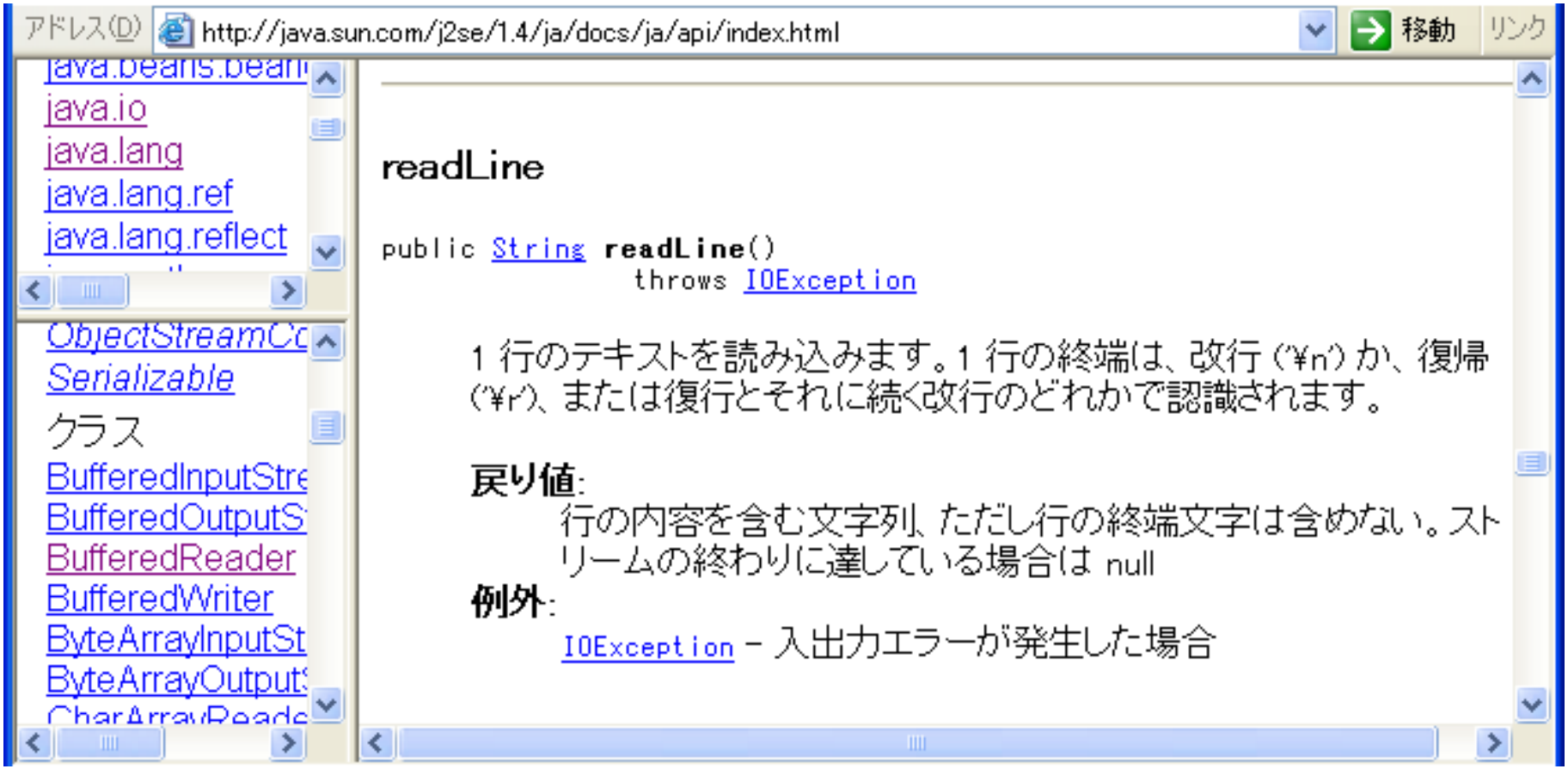
■BufferedReaderクラスのメソッドの概要の中には、すでに使っている、“readLine”のメソッドも見つかります。

■これをクリック（①）してみます。



（8．3．3）メソッドの詳細

■BufferedReaderクラス、“readLine”メソッドの解説が得られます。



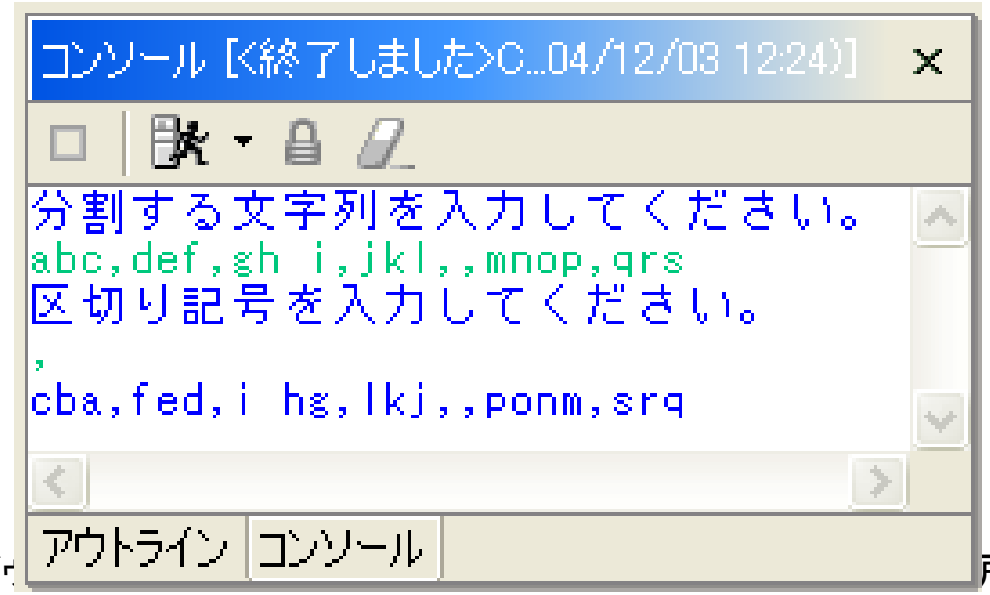
（８．４）ドキュメントの利用と標準クラスの勉強

- 汎用性がある内容で、「こんなJavaのクラスがあったら便利だな」と思うようなクラスは、ほとんど誰かがすでに作っており、利用できるように準備されています。標準クラスには、そのようなクラスで基本的なものすべてが含まれています。
- Javaでは、整備されたオンライン・ドキュメントで利用するクラスの仕様を見ながら、プログラミングを行うスタイルを取ります。
- しかしながら、標準クラスの中でも本当に基本的なものは、その機能について勉強しておく必要があります。
- 以下、スライド（９）では、一番基本的なクラスについて、使い方を勉強します。

（8．5）課題

■課題８－１

- 文字列と区切り記号とを入力して、区切り記号で分かれたそれぞれの部分の文字の順序を逆転させた文字列を出力するプログラムを作成してください。
- “lesson”というパッケージ内にクラスを作ることにします。
- 例外処理は、catch～try～を使ってください。
- ヒント：Stringクラスのメソッドを2つ使います。オンラインマニュアルで調べてください。



（９）基本的な標準クラス

■次のクラスについて、直感的な理解を得るための説明を行っていきます。個々のメソッド等の仕様に関する説明は、基本的にしていません（オンラインドキュメントを参照してください）。

- （９．１）java.lang.StringBufferクラス
- （９．２）ラッパークラス(java.lang)
- （９．３）java.lang.Mathクラス
- （９．４）java.util.ArrayListクラス
- （９．５）java.util.HashMapクラス

（９．１）StringBufferクラス

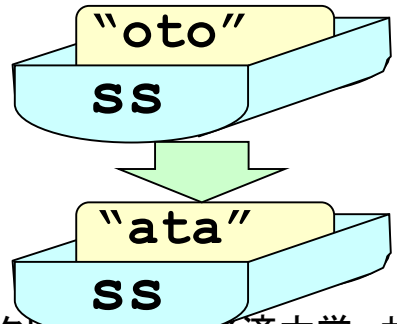
■StringBufferクラスは、Stringクラスと同様に文字列を格納するクラスです。

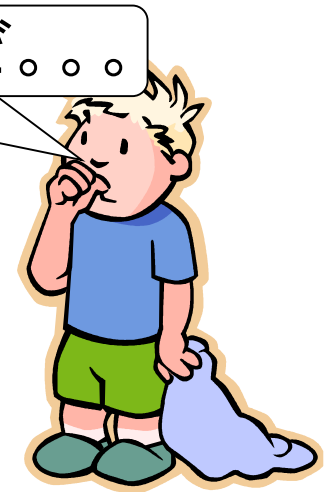
■両者の違いは、“格納した文字列を変更できるか否か”にあります。

- Stringクラス　⇒　変更出来ない。
- StringBufferクラス　⇒　変更出来る。

■「Stringだって変更出来るじゃないか」と思われるかも知れません。どういう意味で「変更出来ない」のかをまず見ていきます。

```
String ss = “oto”;
ss = ss.replace(‘o’,’a’);
// ssには、数”ata”が入る。
```

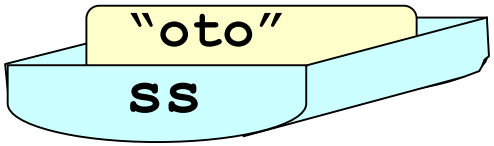




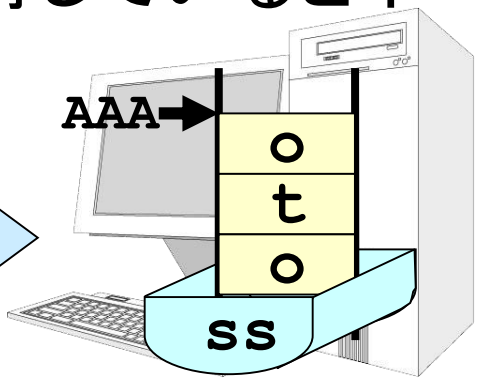
（９．１．１）Stringクラスは変更出来ない

■Stringクラスが文字列を格納しているとき、それぞれの文字がコンピュータ上のメモリ領域を占有しているとイメージしてください。

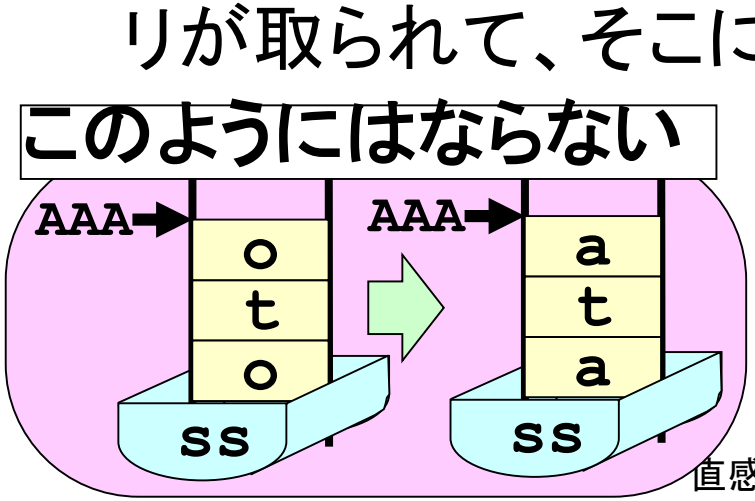
```
String ss = “oto”;
```

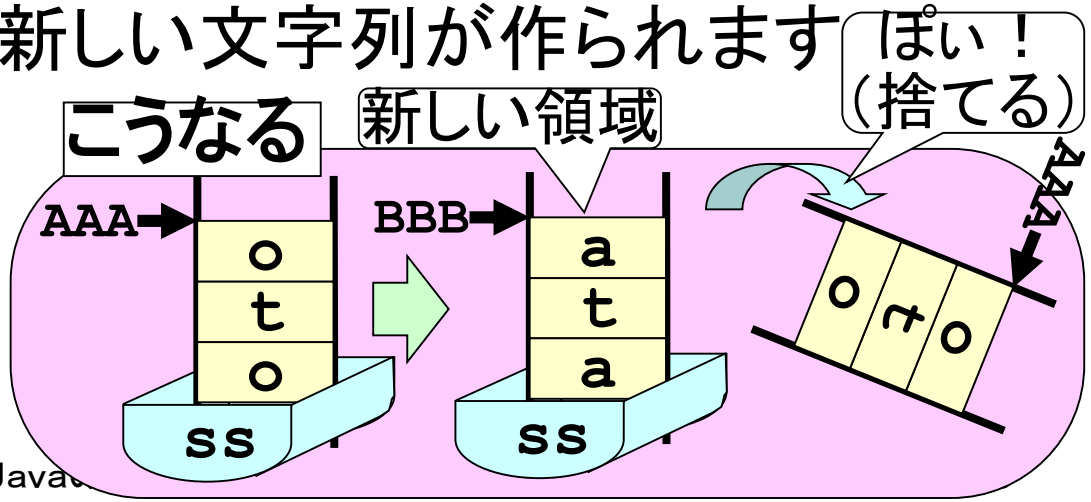


イメージ



■前スライドのように、Stringクラスの文字列が書き換えられた際、メモリが書き換わるのではなく、新たにメモリが取られて、そこに新しい文字列が作られます

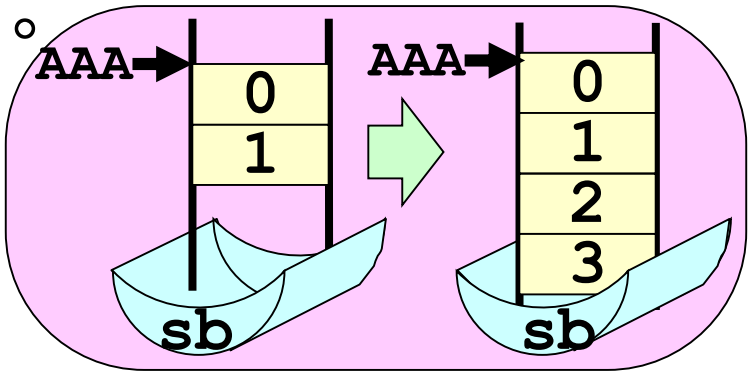




（9．1．2）StringBufferクラスを使うべきとき

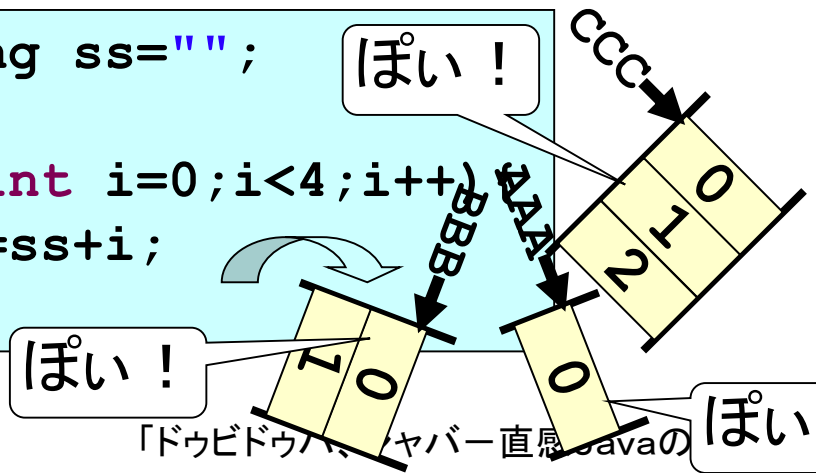
■StringBuffer上の文字列を変更する際には、メモリ上の値を書き換えることになります。

```
StringBuffer sb
= new StringBuffer(“01”);
sb.append(“23”);
// ssには、数”1234”が入る。
```



■変更（例えば末尾への追加）が繰り返される場合、StringBufferを用いて無駄にメモリ領域が捨てられることを防ぎます。

```
String ss="";

for(int i=0;i<4;i++){
  ss=ss+i;
}
```

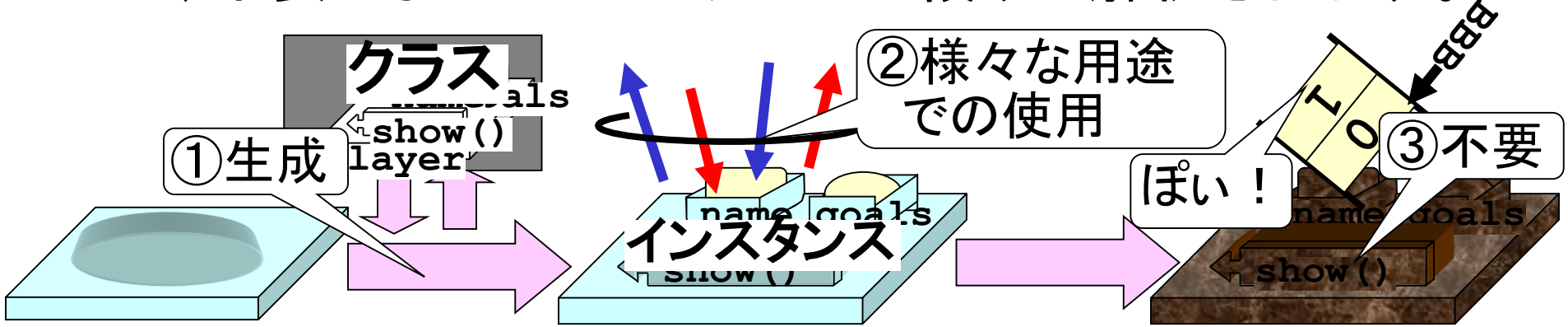



```
StringBuffer sb
=new StringBuffer("");
for(int i=0;i<4;i++){
  sb.append(i);
}
```

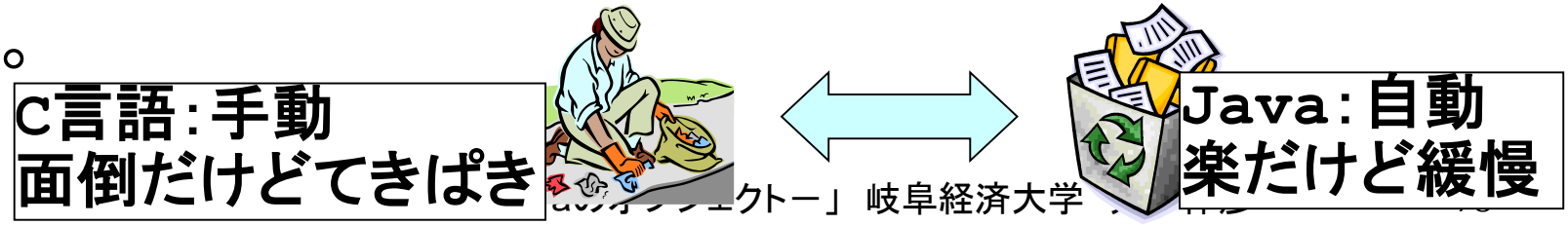
同等の処理だけど、ゴミが出ない。

（９．１．３）ゴミはどこへ行くのか？

■Stringクラスの文字列の書き換え相当で発生するゴミは、不要となったインスタンスの領域と解釈されます。



■Javaでは、ガベージコレクション（garbage collection:ゴミ集め）といって、システムが自動的にこのような領域の解放を行います（CやC++ではプログラマが解放処理をプログラミングしていました）。Javaの自動のゴミ処理は、Cなどの手動のゴミ処理より緩慢（てきぱき片付かない）なので、前スライドのような配慮が必要になります。



（９．１．４）課題

■課題９－１－１

- スライド「Java覚書」の課題９－３を、StringBufferを用いて書き換えてください。

※ 実際には、課題９－３程度の小さなプログラムでは、Stringを用いても、StringBufferを用いても、大差はありません。時間が掛かるような計算量を伴うプログラム、クリティカルな動作を行うオンライン・プログラムなどでは、 StringBufferを用いる配慮が必要となります。

（９．２）ラッパークラス

■ラッパークラスとは、int,char,doubleなどの基本データ型の変数を取り扱うときに便利な道具を提供するクラスです。

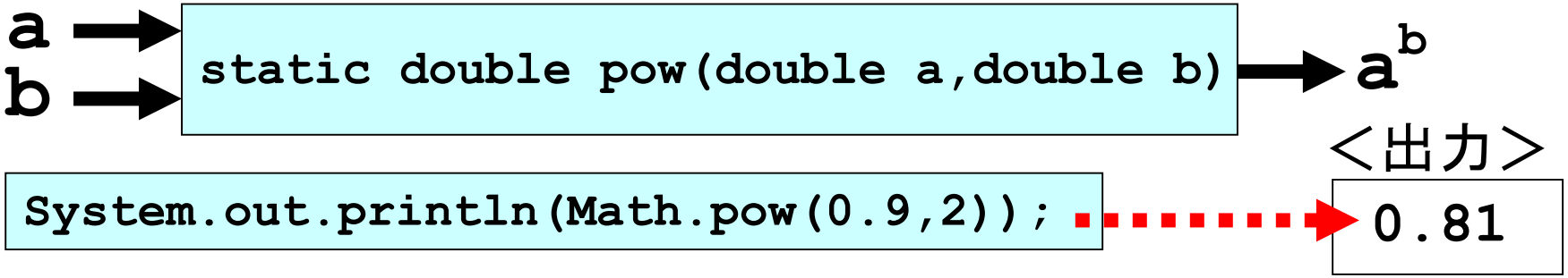
■各基本データ型に対応して、次のようなラッパークラスがあります。

基本データ型	ラッパークラス
boolean	Boolean
char	Character
int	Integer
double	Double

■スライド「Java覚書」（13.3.2）“主な変換”中の変換のいくつかは、上記のラッパークラスを用いて行っています。

（９．３）Mathクラス

■Mathクラスは、数学関係のメソッドを集めたもので、すべてstaticメソッドからなりたっています。すなわち、与えられた引数の値から計算を行って、答えの数値を返す形のメソッドの集まりです。使い方は難しくないはずです。

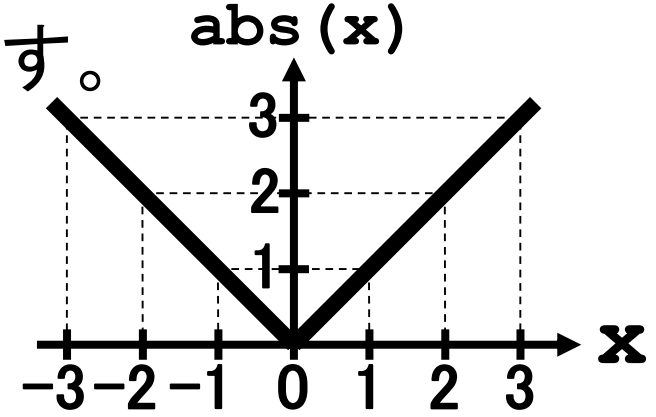


■ここでは、次の２つのメソッドについて、プログラミングでのヒント的な内容を紹介します。

- Math.abs()
- Math.random()

（９．３．１）Math.abs()メソッド

■absメソッドは、絶対値を返す関数です。



x	−3	−2	−1	0	1	2	3
Math.abs(x)	3	2	1	0	1	2	3

■ひし形を文字で描く際に、このabsメソッドを利用すると便利です。

	i(=行)	空白の数	＂＊”の数
*	0	3	1
***	1	2	3
*****	2	1	5
*******	3	0	7
*****	4	1	5
***	5	2	3
*	6	3	1
		abs(3-i)	

absを使って”i”で表すとどうなる？

（９．３．２）Math.random()メソッド

■randomメソッドは、乱数を発生させるメソッドです。

```
static double random()
```

→ **0以上1未満の乱数**

```
for(int i=0;i<5;i++){
  System.out.println(Math.random());
}
```

＜出力＞

```
0.21994744242728792
0.7563056356044705
0.9366330372956264
0.4782882145211 79
0.8513166363922027
```

■randomメソッドを用いて、様々な範囲・値の乱数を作成できます。

```
double x = 10*Math.random();          // 0.0から10.0未満の乱数
int iran = (int)(10*Math.random()); // 0、1、……、9の乱数
int jran = (int)(1+6*Math.random());// 1、2、……、6の乱数
int kran = (int)(2*Math.random());  // 0か1の乱数
```

（９．３．３）課題

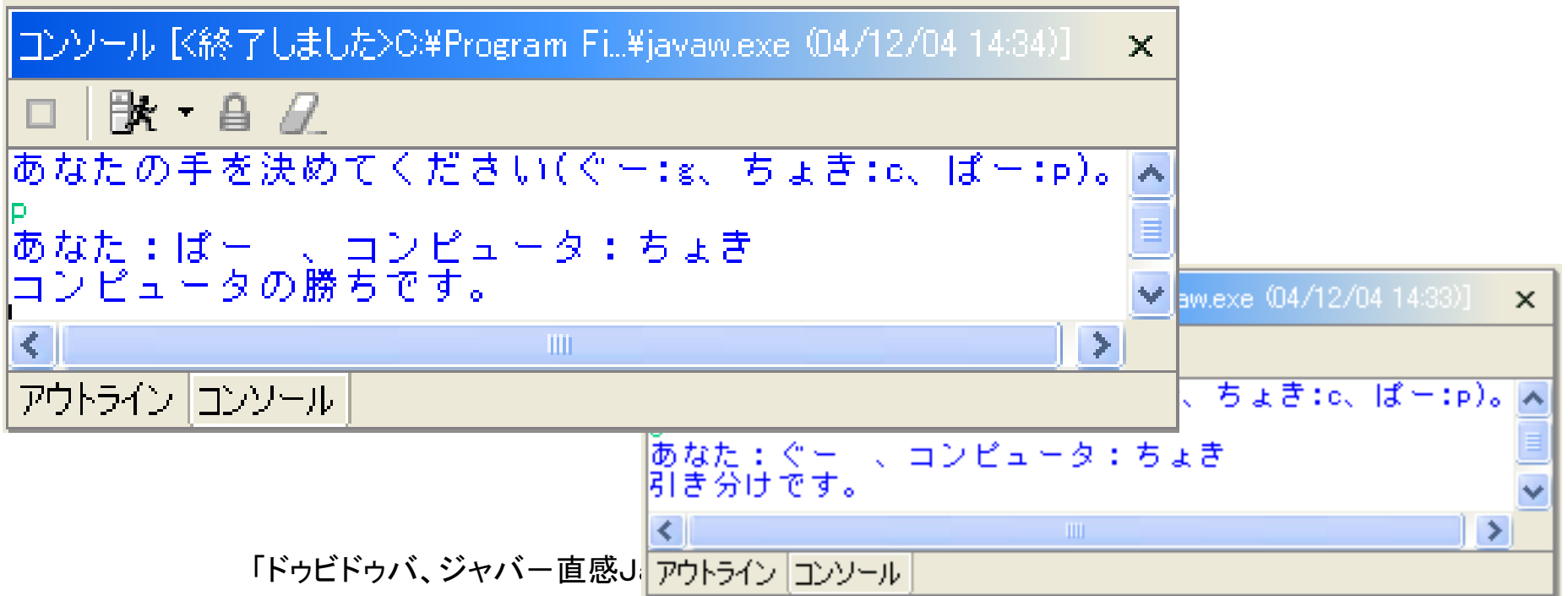
■課題９－３－１

- Math.abs()メソッドを使って、
  記号によるひし形を描く
  プログラムを作成してください。

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

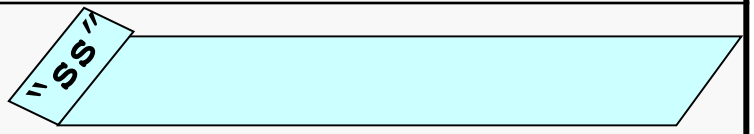
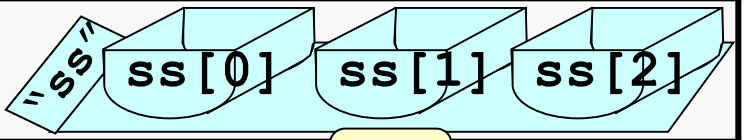
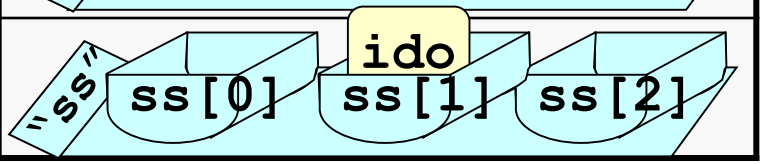
■課題９－３－２

- コンピュータとジャンケンを行うプログラムを作成してください。



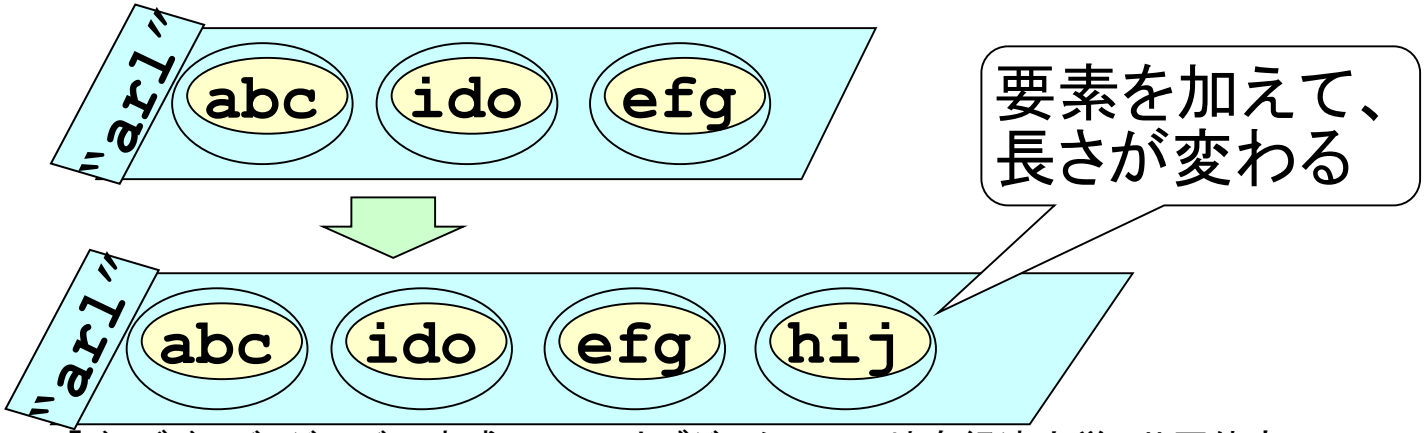
（９．４）java.util.ArrayListクラス

■配列は、使用する前にその長さを決めなければなりません。

記述例	操作跡の変数の内容
`String ss[];`	
`ss[]=new String[3];`	
`ss[1]="ido";`	

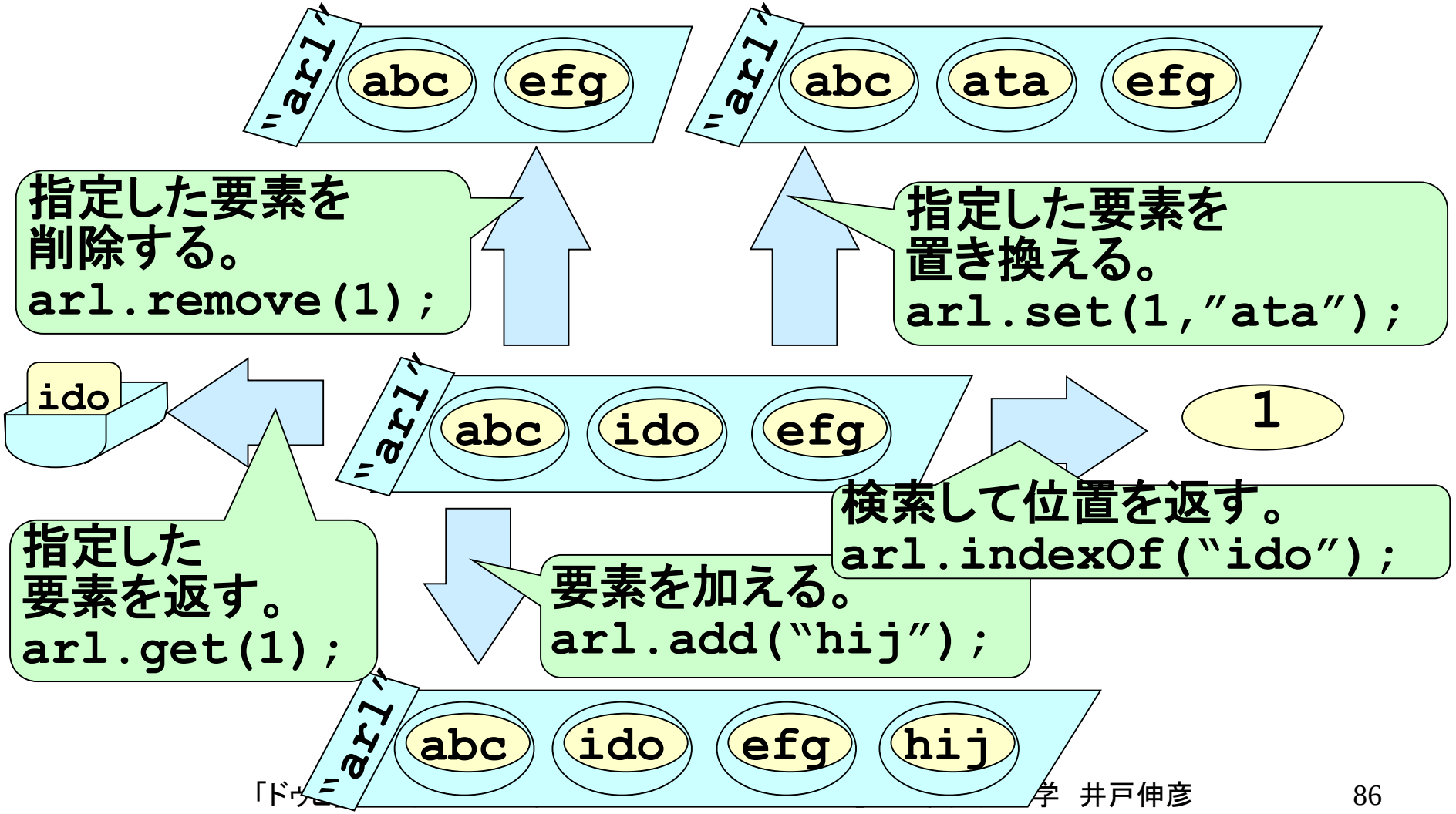
長さは3で決まり、変更不可。

■ArrayListクラスは、サイズを変更出来る配列です。



（９．４．１）ArrayListへの操作

■他の操作についても、オンラインマニュアルで確認してください。



（９．４．２）どのような値を保持するのか？

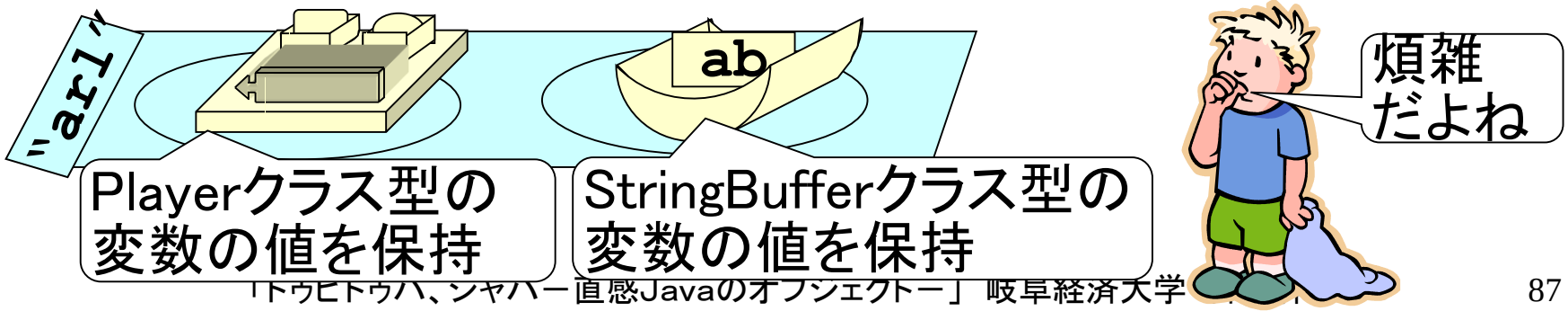
■ArrayListクラスは、インスタンス生成時に、特定の型（基本データ型、クラス型）を保持するように宣言することが出来ます。

“Player”クラスのインスタンスを保持。

```
ArrayList <Player> arl = new ArrayList<Player>();
```

■型を指定しないと、あらゆるクラス型（“Object”クラスの型）の変数の値を保持しますが、そのような使い方はしないようにしてください。

```
ArrayList arl=new ArrayList();
Player pl = new Player(); arl.add(pl);
StringBuffer sb = new StringBuffer(“ab”); arl.add(sb);
```



（９．４．３）整数値を保持するArrayList

■ArrayListに基本データ型である整数（int）を保持する場合は、ラッパークラス（スライド(9.2)）”**Integer**”を指定します。

```
ArrayList <Integer> arl=new ArrayList<Integer>();
arl.add(1);
```

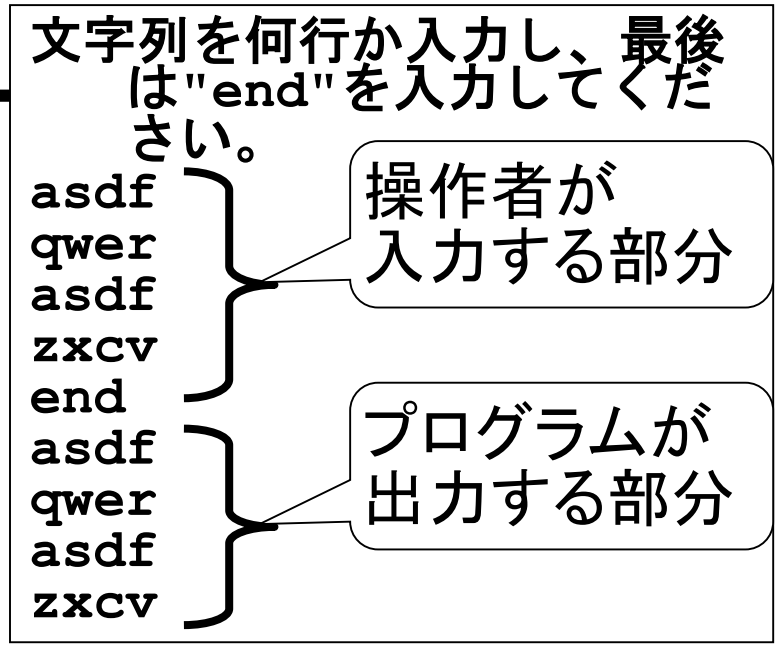
■他の基本データ型のArrayListを作る際も、同様にラッパークラスを指定します。

```
ArrayList <Double> arl_1=new ArrayList<Double>();
arl_1.add(3.14);

ArrayList <Character> arl_1=new ArrayList<Character>();
arl_1.add('a');

ArrayList <Boolean> arl_1=new ArrayList<Boolean>();
arl_1.add(true);
```

（９．４．５）課題

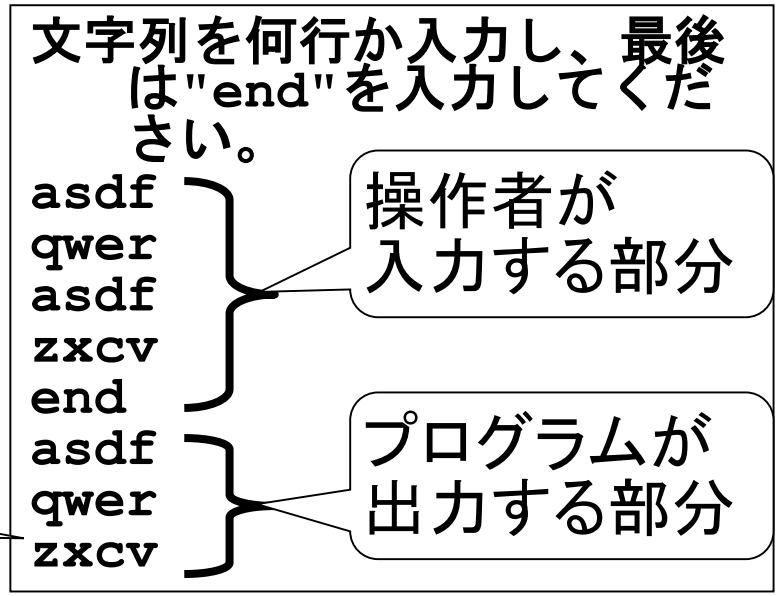
■課題９－４－１

- 文字列をコンソールから１行ずつ入力し、“end”が入力された時点でそれまで入力した行を出力するプログラムを作成してください。



■課題９－４－２

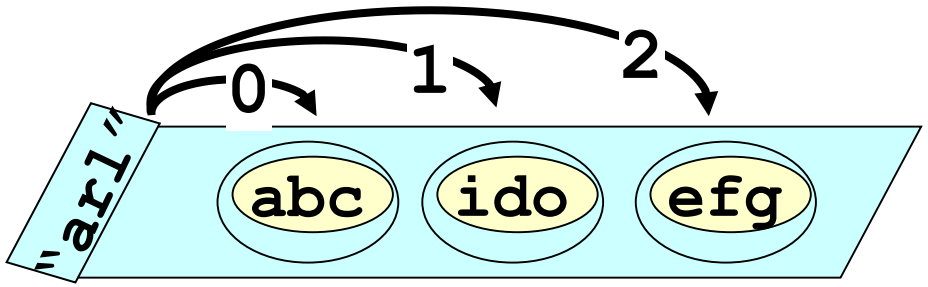
- 課題９－４－１のプログラムで、入力した行のうち同じ文字列の行は１回だけ出力するよう、プログラムを改造してください。

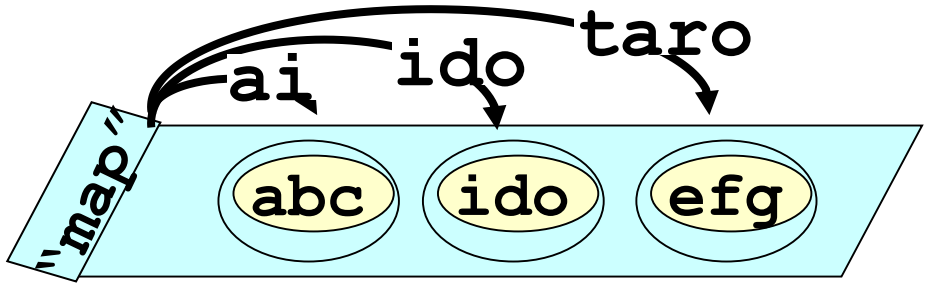


２回目の“asdf”は出力されない

（９．５）java.util.HashMapクラス

- ArrayListは、基本データ型の配列と同様に、非負整数（0,1,2,...）で索引するものでした。
- HashMapクラスは、文字列などの引数で、値を索引することができます。
- すわなち、HashMapは2列の表にて、キー(key)となる1列目の値から2列目の値を求めることが出来る表の機能を果たします。



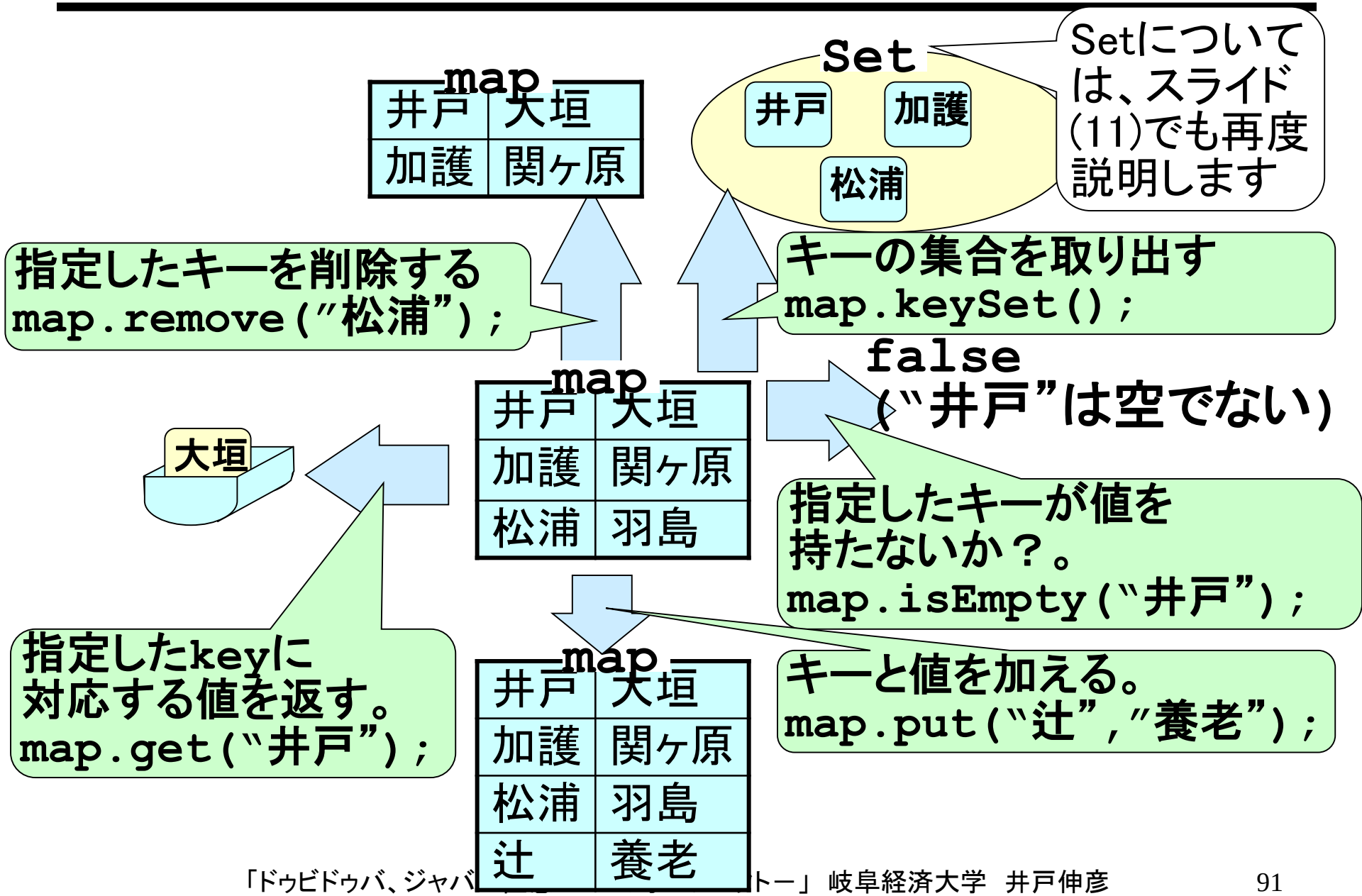


①keyとなる
こちらの列から

②こちらの列
の値を求める

井戸	大垣
加護	関ヶ原
辻	養老

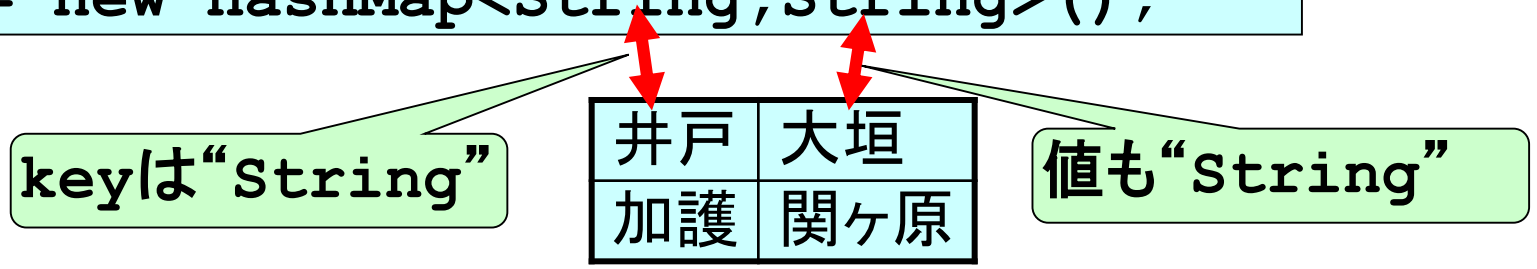
（９．５．１）HashMapへの操作



（９．５．２）どのような値を保持するのか？

■ArrayListクラスと同様に、インスタンス生成時に、特定の型（基本データ型、クラス型）を保持するように宣言します。

```
 HashMap <String,String> map
= new HashMap<String,String>();
```




```
map.put(“井戸”,”大垣”);
```

■ArrayListクラスと同様に、型を指定しない使い方はしないようにします。

（９．５．３）一覧の取り出し　－１－

■HashMapは配列ではありませんから、その一覧は次のようにします。

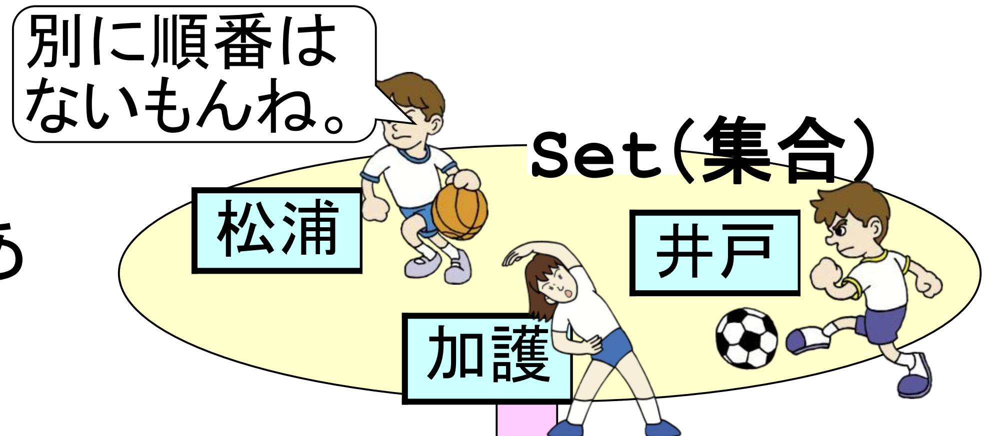
```
 1:HashMap <String,String> map
   = new HashMap<String,String>();
 2:map.put("井戸","大垣");
 3:map.put("加護","関ヶ原");
 4:map.put("辻","養老");
 5:Set<String> mapSet=map.keySet();
 6:Iterator <String>iterator
   =mapSet.iterator();
 7:while(iterator.hasNext()){
 8:   String key = iterator.next();
 9:   System.out.println(key+"="+map.get(key));
10:}
```

＜出力＞

```
辻 = 養老
加護 = 関ヶ原
井戸 = 大垣
```

（9．5．4）一覧の取り出し　－4－

■HashMap(map)から取り出した、Set(集合、mapSet)には、順序がありません。




```
6:Iterator <String>iterator=mapSet.iterator();
```

■6行め：配列のように引数で順次処理をすることが出来ないので、Iterator(繰り返し) を作って、集合の要素を順次取り出します。

（9．5．5）一覧の取り出し　－3－

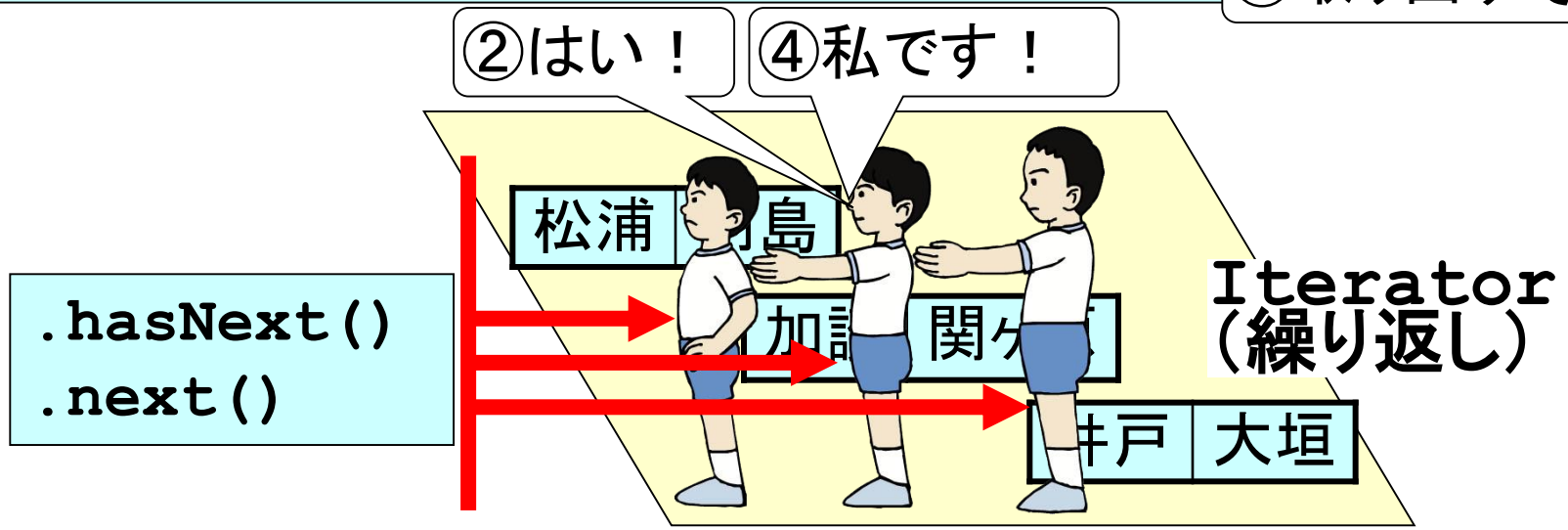
■7，8，9，10行め：Iteratorの次のメソッドを使って、集合の要素について順次処理します。

- “**hasNext()**”メソッド　⇒　次の要素があるか否かの判定
- “**next()**”メソッド　⇒　次の要素を取り出す。

```
 7:while(iterator.hasNext()){
 8:   String key = iterator.next();
10:}
```

①次、居るか？

③取り出すぞ！

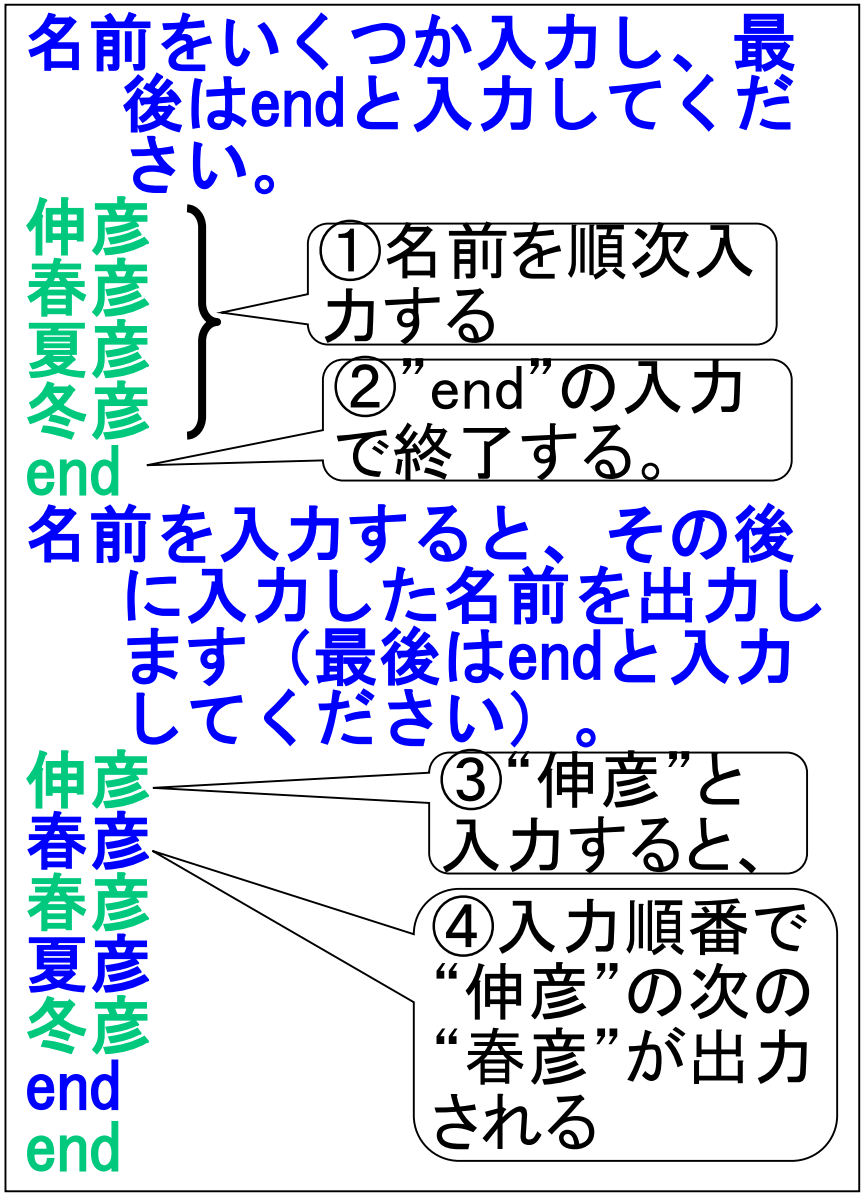


（９．５．７）課題

■課題９－５－１

次のようなプログラムを作成してください。

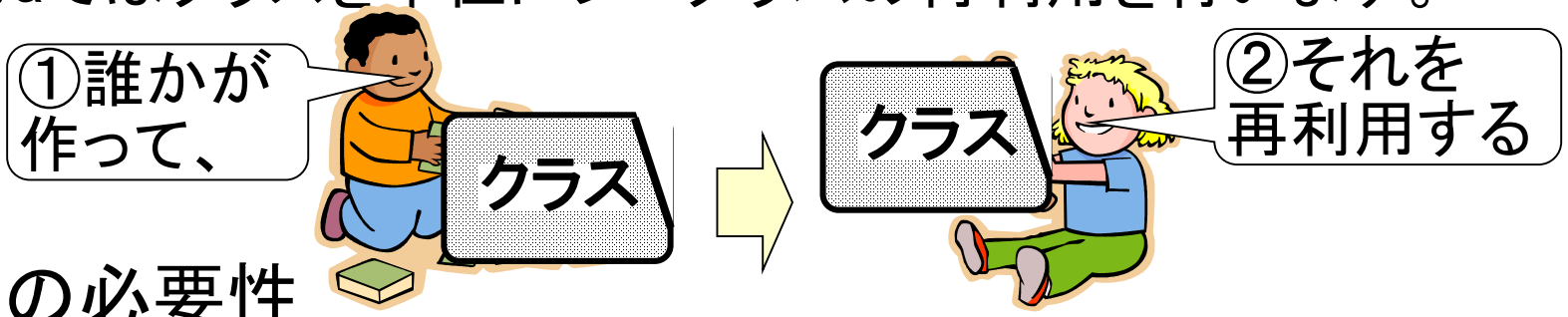
- プログラムでは最初、いくつかの名前を入力します（①）。
- “end”が入力されると名前の入力を終了します（②）。
- 次に、今まで入力した名前を入力する（③）と、先の入力順序で次にあたる名前（④）が出力されます。これを繰り返します。
- 最後も“end”が入力されると、プログラムの実行を終了します。



（10）継承（Inheritance）

■プログラムの再利用

- オブジェクト指向プログラミングの主な目的のひとつに、プログラムの再利用（ある目的に作成したものを、別の目的でも利用する）を効率的に行うことがあります。
- Javaではクラスを単位にプログラムの再利用を行います。

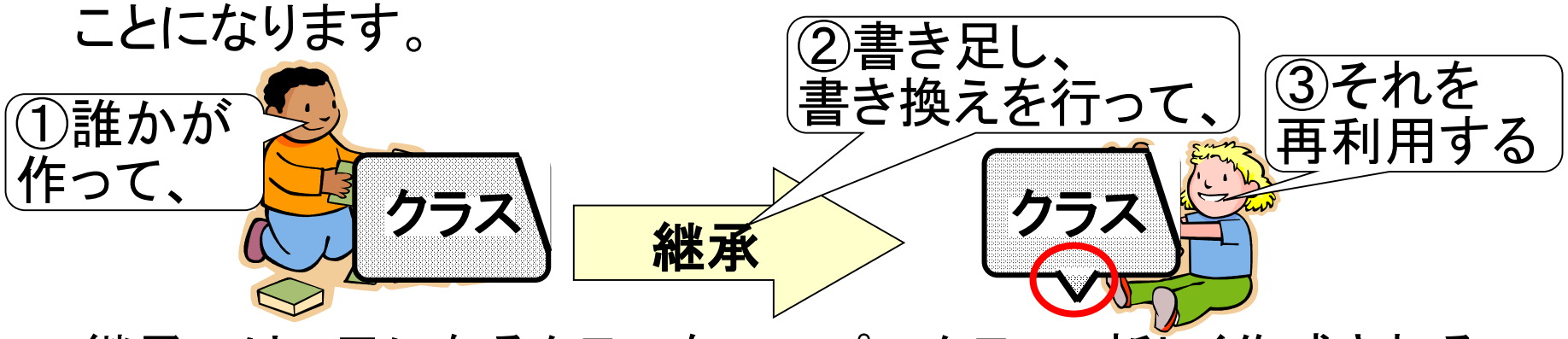


■継承の必要性

- 再利用したいのだけれども、少し仕様が異なっている場合、継承により再利用が可能となります。

（10．1）書き足し、書き換え

■継承では、元になるクラスを書き足したり、書き換えたりして使うことになります。

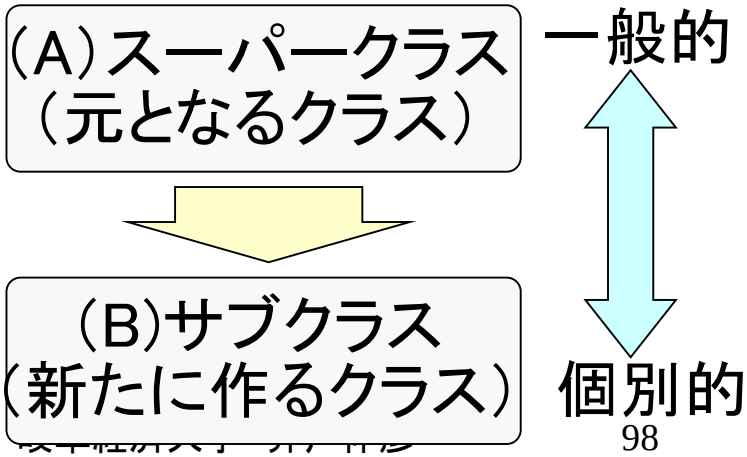


■継承では、元になるクラスをスーパークラス、新しく作成されるクラスをサブクラスと言います。

■スーパークラス“A”と、そのサブクラス“B”との関係は、次のような言い方で表されます。

- BはAを拡張している。
- BはAを継承している。
- AはBの汎化である。

■あるクラスが継承できるスーパークラスはひとつだけです。

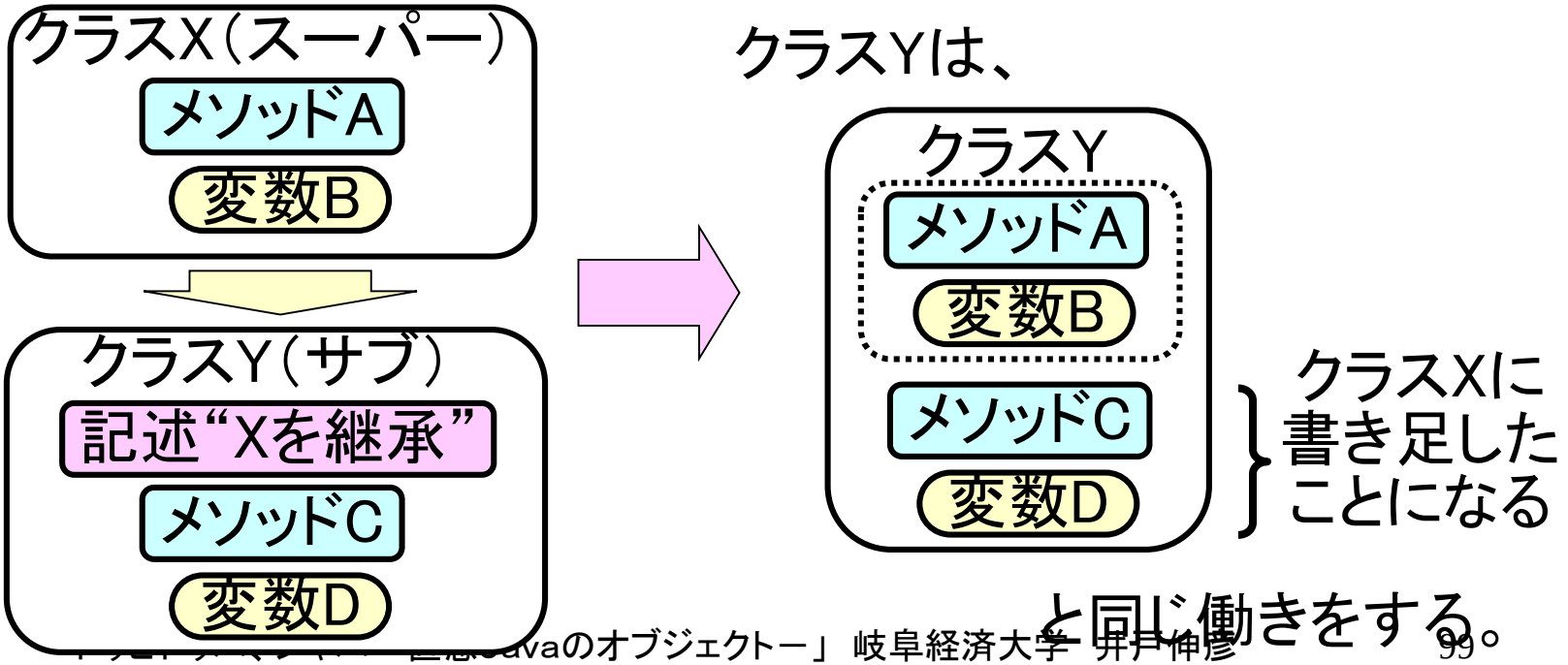


（10．2．1）書き足し

■クラスは、フィールド変数とメソッドとから成ります。

■クラスXをクラスYが継承しているとします（すなわち、クラスXがスーパークラス、クラスYがサブクラスです）。

■このとき、クラスXの変数とメソッドとはクラスYにも記述されていることと同じになります。クラスYに変数とメソッドを作成すれば、それは書き足されたことになります。



（10．2．2）書き足しのコーディング

■Javaのコーディングでは、“**extends**(拡張する)”の予約語を用いて、継承することの記述を行います。

```
# クラスX(スーパークラス)
class ClassX{
  int b;
  void methodA(){
        :
  }
}
```

```
#クラスY(サブクラス)
class ClassY extends ClassX{
  int d;
  void methodC(){
        :
  }
}
```

“classXを継承する”という記述

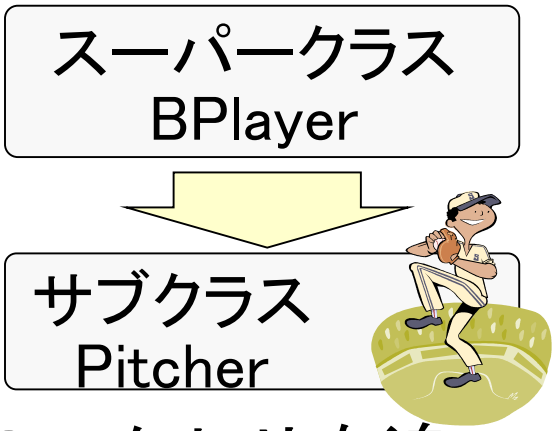
クラスclassYは、

```
#クラスY
class ClassY{
  int b;
  int d;
  void methodA(){
        :
  }
  void methodC(){
        :
  }
}
```

と同じ働きをする。

（10．2．3）Pitcher(投手)クラス

■パッケージbaseballに、投手のクラス“Pitcher”を、BPlayerクラスを継承して作成してみます。



■Pitcherクラスには、投球回数（innings）というint型のフィールドを設け、これへのアクセサを追加します。

BPlayerクラスを拡張

```
package baseball;
public class Pitcher extends BPlayer {
  private int inning; // 投球回数
  public void setInning(int inning) {
    this.inning = inning;
  }
  public int getInning() {
    return inning;
  }
}
```

（10．2．4）Pitcherクラスの利用

■生成したPitcherインスタンスでは、次のメソッドが利用できます。

- ①Pitcherクラスで定義したメソッド（setInnings,getInnings）
- ②継承元のBplayerクラスで定義したメソッド

```
package baseball;
public class BaseballTeam10 {
  public static void main(String[] args) {
    Pitcher pc = new Pitcher();
    pc.setName("野茂");
    pc.setPosition("投手");
    pc.setRuns(0);
    pc.setInning(214);
    pc.show();
    System.out.println("投球回数="
                       +pc.getInnings());
  }
}
```

②
①
②
①

＜出力＞

```
野茂投手0
投球回数=214
```

（10．2．5）コンストラクタの継承

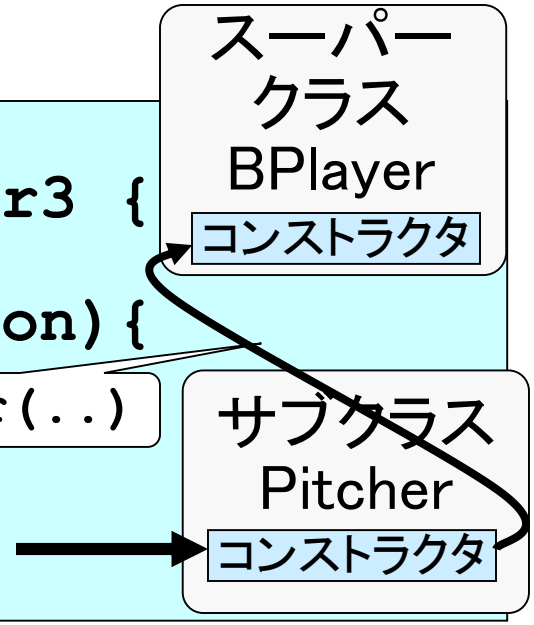
■コンストラクタの継承は次のとおりとなります。

- 引数の無し　⇒　サブクラスでも自動的に実行される
- 引数の有り⇒　継承されない

■Pitcherクラスと同じ引数のあるコンストラクタをBPlayerクラスで使用したい場合には、スーパークラスのコンストラクタを現す“super”を用いて、次のようにコンストラクタを記述します。

```
package baseball;
public class Pitcher extends BPlayer3 {
    :
   Pitcher(String name,String position){
     super(name,position);
   }
    :
}
```

スーパークラスであるBPlayerクラスのコンストラクタを示す



（10．2．6）課題

■課題10－2－1

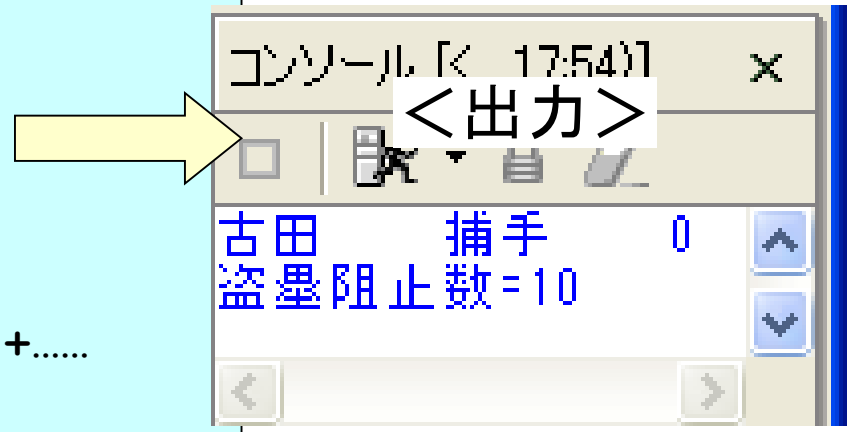
- スライド(10.2.3)～(10.2.5)のように、Pitcherクラスを作成して呼び出してください。

（10．2．7）課題

■課題10－2－2

- 次のような、Catcher(キャッチャー)クラスを作成してください。
  - ◆Bplayer3を継承する。
  - ◆throwOut(盗塁阻止)数をフィールド変数に持ち、アクセサを持つ。
- Catcherクラスを使って、次のような出力を得るプログラムを作成してください。

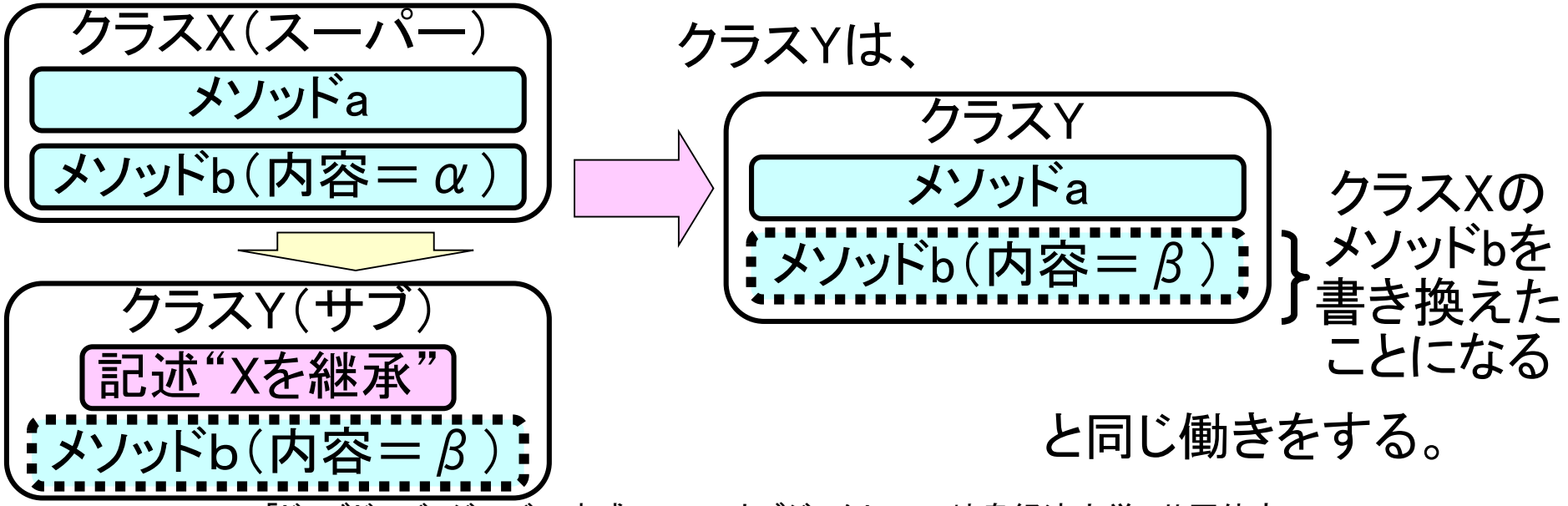
```
package baseball;
public class BaseballTeam3 {
  public static void main(String[] args) {
    Catcher ct = new Catcher();
    ct.setName("古田");
    ct.setPosition("捕手");
    ct.show();
    //盗塁阻止数を10に設定する
        :
    //盗塁阻止数を出力する
    //System.out.println("盗塁阻止="+.....
        :
  }
}
```



（10．3．1）書き換え

■元となるクラスのメソッドを書き換えたい場合、オーバーライド（override）という方法を取ります。

■スーパークラスのクラスXのメソッドを書き換えたい場合、同じ名前と引数のメソッド（下図ではb）をサブクラスのクラスYに記述します。すると、クラスYではクラスXのメソッドbは使われず、クラスYに記述したメッソッドbが使われます。



（10．3．2）書き換えのコーディング

■サブクラスのclassBでは、スーパークラスのclassAと同じメソッド“**methodB**”を記述します。

```
# クラスX(スーパークラス)
class ClassX{
   int c;
   void methodA(){
          :
   }
   void methodB(){
     # 私は、X、X、X、X
   }
}
```

オーバーライドされるメソッド

```
#クラスY(サブクラス)
class ClassY extends ClassX{
   void methodB(){
     # 私は、Y、Y、Y、Y
   }
}
```

オーバーライドするメソッド

クラスclassYは、

```
#クラスY
class ClassY{
   int c;
   void methodA(){
          :
   }
   void methodB(){
     #私は、Y、Y、Y、Y
   }
}
```

と同じ働きをする。

（10．3．3）Keeper(キーパー)クラス

■パッケージfootballに、キーパーのクラス“Keeper”を、Playerクラスを継承して作成してみます。

■Keeperクラスには、ゴール数は失点と捉えて、0以下の数値しか許容しないことにし、setGoalsメソッドをオーバーライドします。

スーパークラス
Player

サブクラス
Keeper

Playerクラスを拡張

```
package football;
public class Keeper extends Player {
  public void setGoals(int goals){
    if(goals>0){
      this.goals = 0;
    }else{
      this.goals = goals;
    }
  }
}
```

（10．3．4）Keeperクラスの利用

■生成したKeeperインスタンスでは、Playerインスタントと同様にsetGoalsメソッドが利用出来ますが、その内容はオーバーライドしたもの（0以下の値のみを許容するもの）となっています。

```
package football;
public class FootballTeam3 {
  public static void main(String[] args) {
    Keeper kp = new Keeper();
    kp.setName("楢崎");
    kp.setGoals(-3);
    Player pl = new Player();
    pl.setName("久保");
    pl.setGoals(-3);
    kp.show();
    pl.show();
  }
}
```

Keeperクラスでも
setGoalsは、
同じように使用できる

＜出力＞

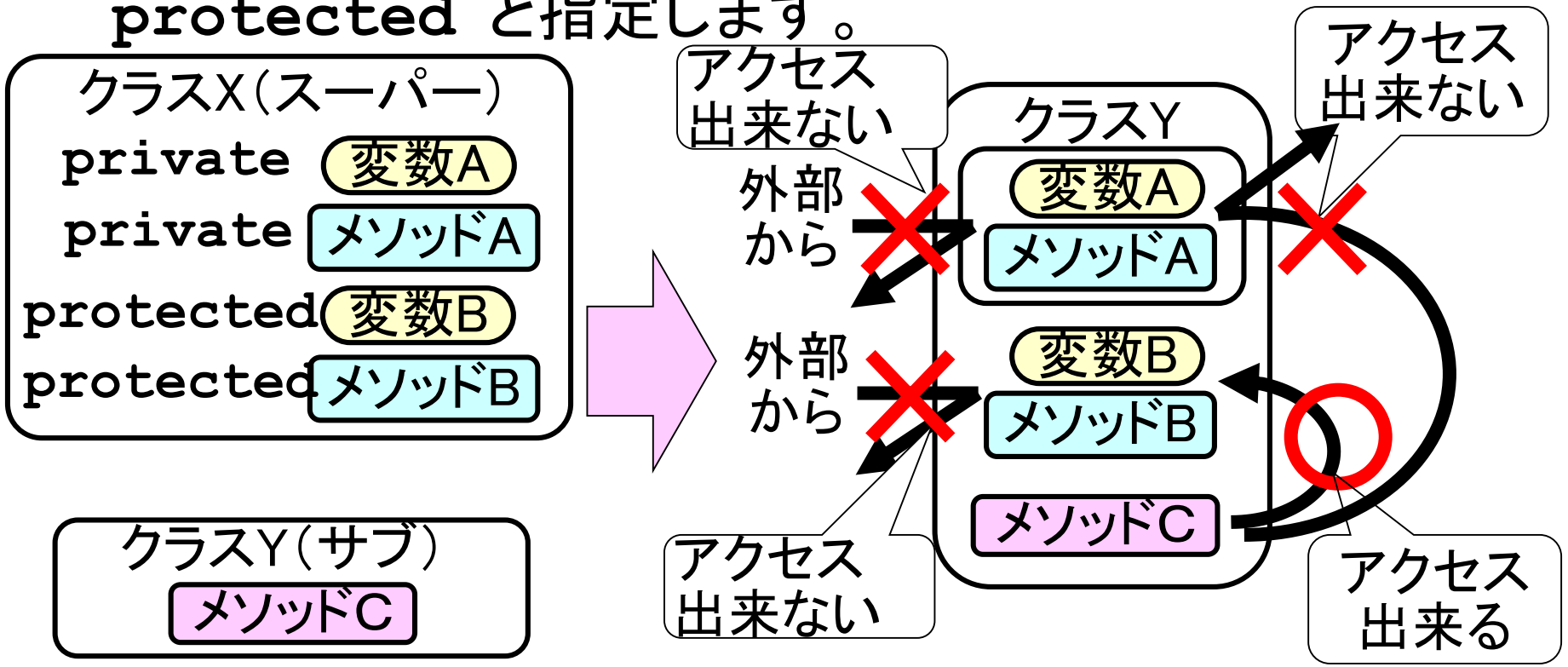
```
楢崎 -3
久保  0
```

Keeperクラスの
setGoalsでは、
負の値が設定される

（10．3．5）protected

■スーパークラスの変数・メソッドに“private”の指定があると、カプセル化によりサブクラスではこれに直接アクセスできません（変数自体は存在します）。

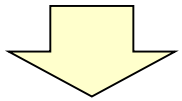
■このため、継承されたクラスでアクセスする場合には、“protected”と指定します。



（10．3．6）Playerクラスの変更

■スライド（10.3.3）（10.3.4）のプログラムが動くためには、次のようにPlayerクラスを変更します。

```
class Player {
  private Stirng name;
  private int goals;
        :
```



```
class Player {
  private Stirng name;
  protected int goals;
        :
```

（10．3．7）スーパークラスのメソッドを呼ぶ

■オーバーライドしたメソッド内で、スーパークラスの該メソッドを呼び出す場合、次のようにします。

```
package football;
public class Keeper extends Player {
      :
   public void show(){
      System.out.println("キーパー");
      super.show();
   }
}
```

スーパークラスの
showを呼び出す

（10．3．8）課題

■課題10－3－1

- スライド(10.3.3)～(10.3.6)のように、Keeperクラスを作成して呼び出してください。

■課題10－3－2

- 課題10－2－2で作成したCatcherクラスを、次のように変更して、Catcher2クラスをください。
  - ◆showメソッドでは、BPlayerと同じ出力以外に、次の出力を行う。
    - ロ盗塁阻止を表示する。
    - ロ盗塁素子数が得点より多い場合、“守備の人”と表示する。

（次スライドへ続く）

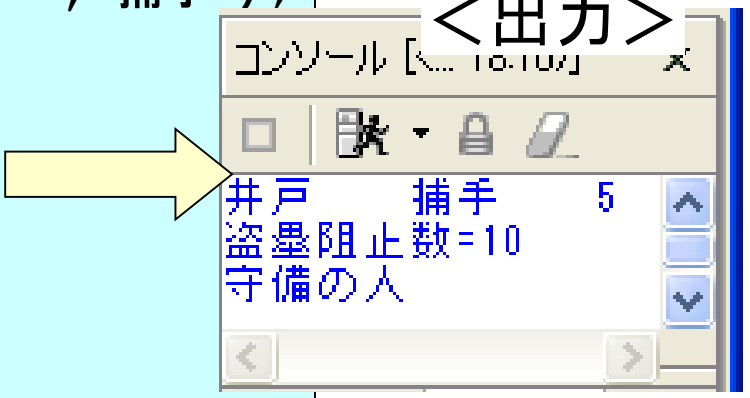
（10．3．8）課題

（前スライドより続く）

- ◆BPlayerで作成した2引数コンストラクタと同じ機能のコンストラクタを持つ。このコンストラクタでは、ポジションについても捕手と設定する。
- 上記の変更による機能が分かるように、Catcher2クラスを起動してみてください。

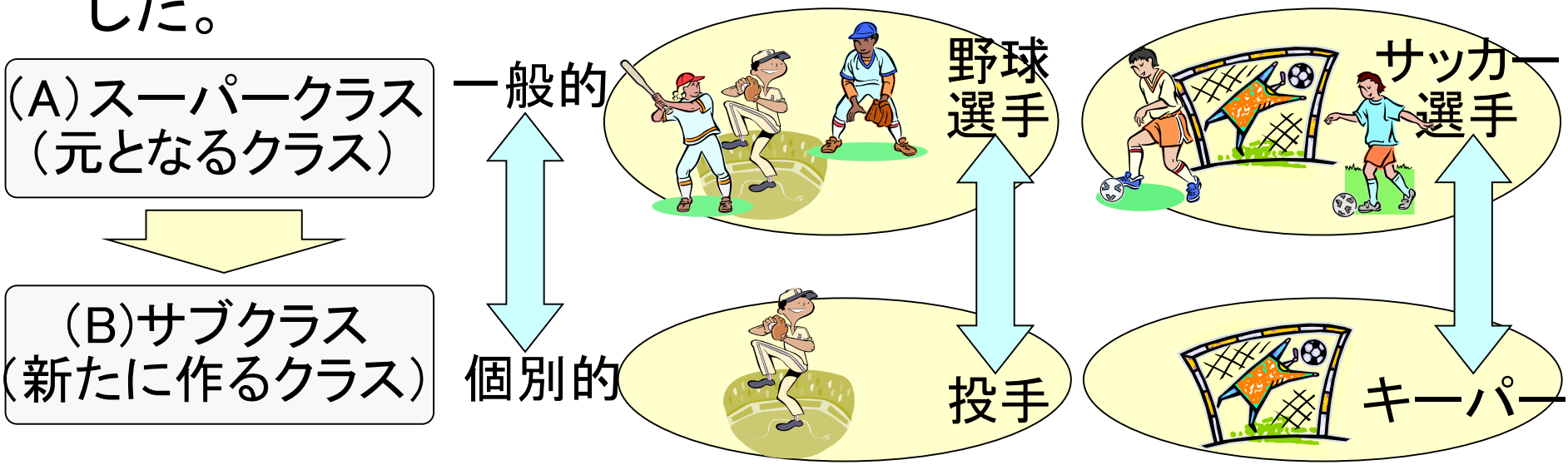
```
package baseball;
public class BaseballTeam1032 {
  public static void main(String[] args) {
    Catcher2 ct = new Catcher2("井戸","捕手");
    //得点(runs)を5に設定する
    ct.setRuns(5);
    //盗塁阻止数を10に設定する
    ct.setCaughtStealing(10);
    ct.show();
  }
}
```

＜出力＞



（10．4）どのような場合に継承を使うか？

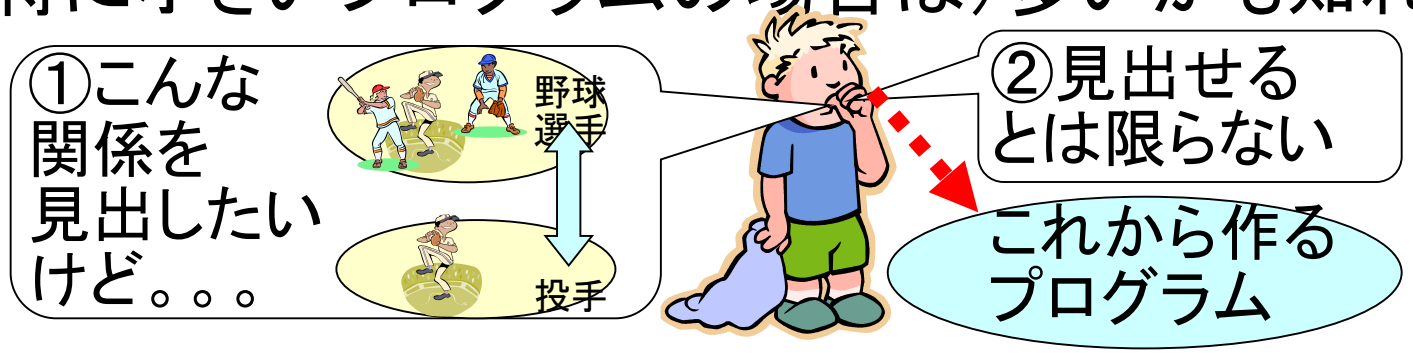
■スライド(10.2)(10.3)では、野球選手に対して投手、サッカー選手に対してキーパーというように、より個別化した内容の“もの”を定義する際に、継承を利用して来ました。



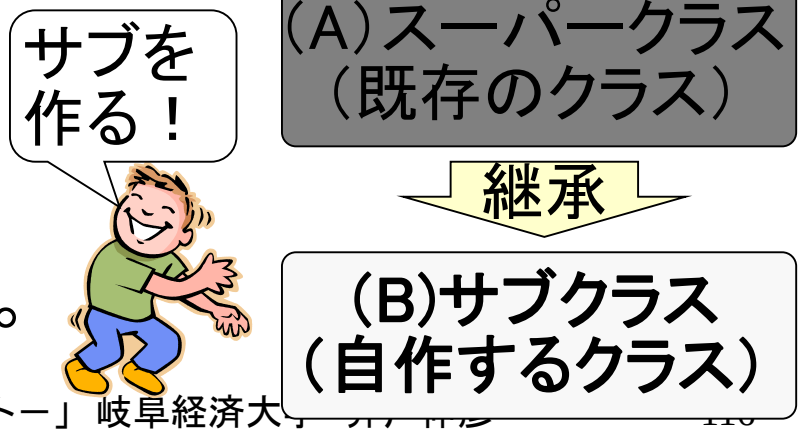
■作成するプログラム中に、このような関係にある“もの”を見出せる場合、それは継承によるモデル化がうまくいっていると言えます。

（10．4．1）継承を使うことは少ないか？

■プログラムを作成する際、前スライドのようにうまく継承のモデル化が出来る“もの”が見出せるとは限りません。むしろ、そのような“もの”が見出せない場合のほうが（特に小さいプログラムの場合は）多いかも知れません。



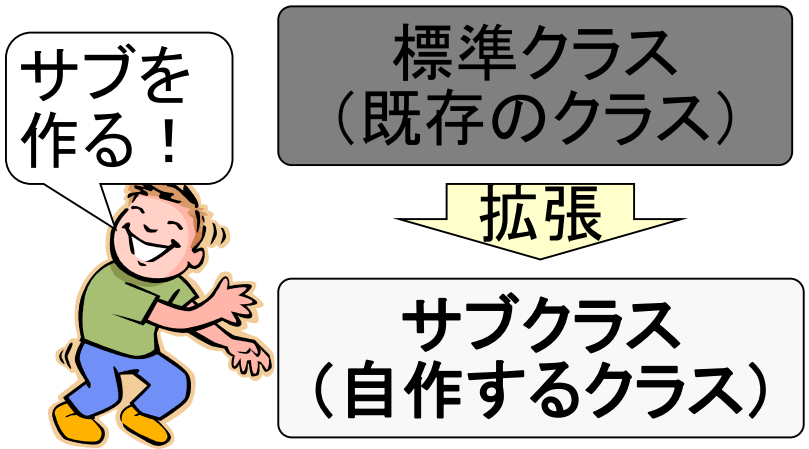
■それではあまり継承は使わないかというと、そうではありません。スーパークラスは既存のプログラムを用い、サブクラスだけ作るといった方法で、継承はよく使われます。



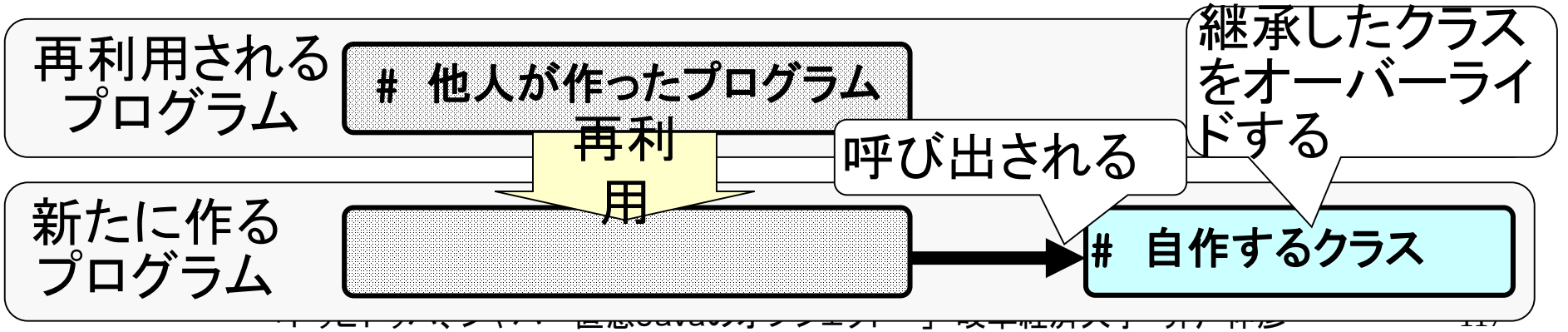
（10．4．2）2つの場合

■前スライドに記したサブクラスのみを作るやり方で継承を使用する代表的な例として、次の2つを考えます。

（1）一般的な機能を持つ
標準クラスを拡張して
利用する（スライド(10.5)）。



（2）プログラムの部品化の仕組みの中で、継承して使うことが前提とされたクラスを利用する（スライド(10.6)）。



（10．4．3）デザイン・パターン

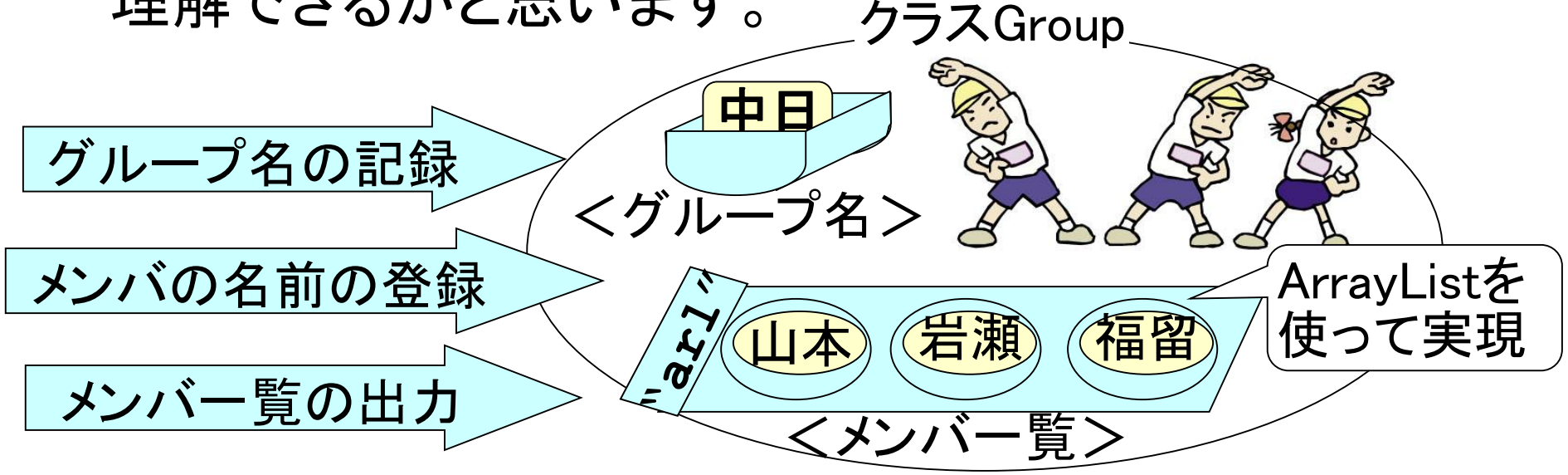
- 本スライドは、入門のレベルを対象として、Javaによるオブジェクト指向プログラミングを説明しています。
- このため、継承などのオブジェクト指向プログラミングの機能を巧みに利用する方法については、扱っていません。
- 「オブジェクト指向プログラミングの機能を巧みに利用する方法」については、“デザイン・パターン”という名でその類型化が提案されています。
- このスライドの勉強を終えられた方は、“デザイン・パターン”について勉強されることをお勧めします。

（10．5）グループの名簿のクラスを作る

■例として、グループの名簿を管理するクラス（Group）を作ってみます。次のような用件を満たすクラスとします。

- 任意の人数のメンバの名前を登録できる。
- グループ名を記録できる。
- グループ名とメンバの一覧を出力するメソッドがある。

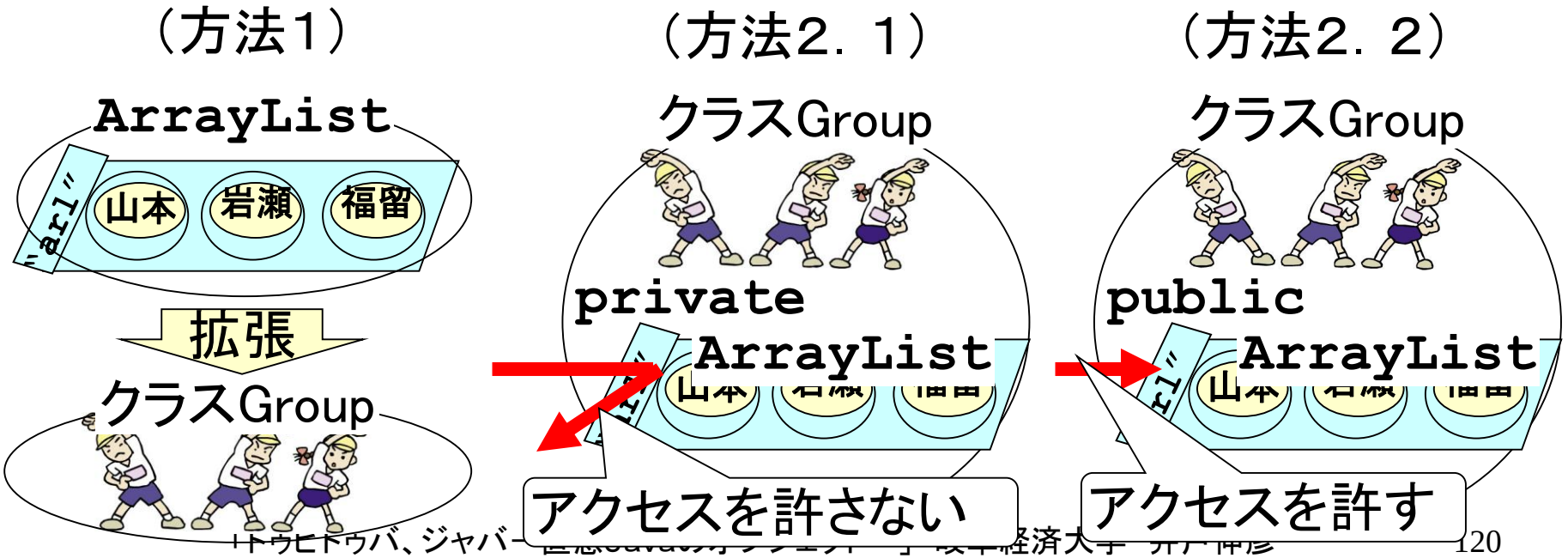
■任意の人数のメンバを記憶するために、標準クラスの“**ArrayList**”（スライド(9.4)）を利用すればよいことは理解できるかと思います。



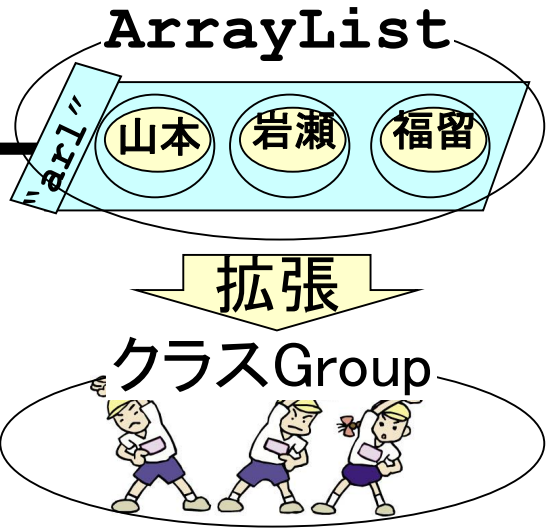
（10．5．1）3つの方法

■前スライドのようなクラス“Group”を次の3とおりの作り方で作成してみます。

- （方法1）“**ArrayList**”を継承してクラスを作成する。
- （方法2）“**ArrayList**”のクラス型変数を、フィールド変数としたクラスを作成する
  - ◆(方法2．1)フィールド変数をprivateとする
  - ◆(方法2．2)フィールド変数をpublicとする



（10．5．2）方法1

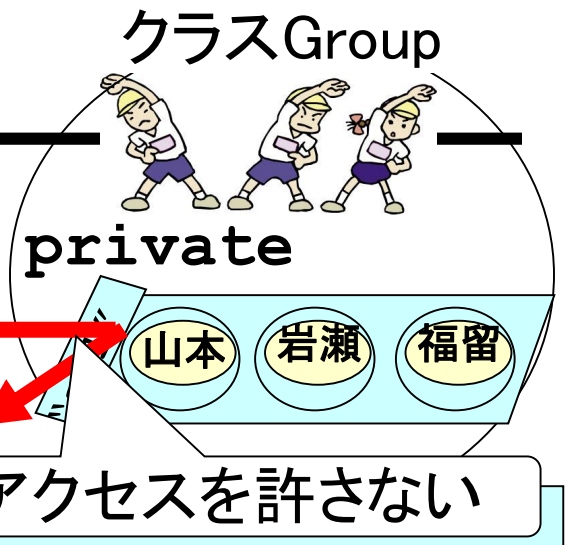


■クラスArrayListを継承した場合、メンバを登録するメソッドは、ArrayListの“add”を用いれば良いので、特に記述する必要はありません。

```
import java.util.ArrayList;
public class Group extends ArrayList<String> {
  private String groupName;
  public void show(){
    System.out.println("*"+this.groupName+"*");
    for(int i=0;i<this.size();i++){
      System.out.println(this.get(i));
    }
  }
  public void setGroupName(String groupName) {
    this.groupName = groupName;
  }
}
```

（10．5．3）方法2．1

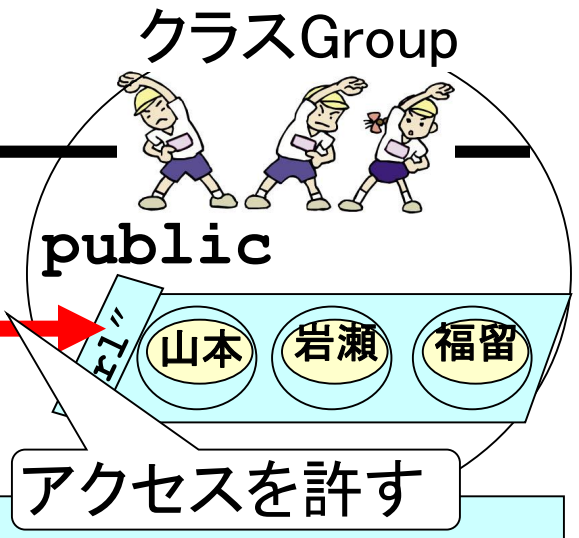
■クラスArrayListをprivateのフィールドで宣言（下では“menbers”）した場合、メンバを登録する　メソッドを記述する必要があります。




```
import java.util.ArrayList;
public class Group {
  private ArrayList<String>members = new ArrayList <String>();
  private String groupName;
  public void add(String member){
    members.add(member);
  }
  public void show(){
    System.out.println("*"+this.groupName+"*");
    for(int i=0;i<members.size();i++){
      System.out.println(members.get(i));
    }
  }
  public void setGroupName(String groupName) {
    this.groupName = groupName;
  }
}
```

（10．5．4）方法2．2

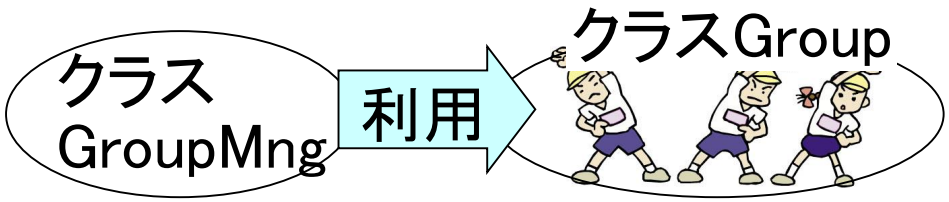
■クラスArrayListをpublicのフィールドで宣言（下では“menbers”）した場合、メンバの登録は該フィールドを直接参照することになります（次スライド）。



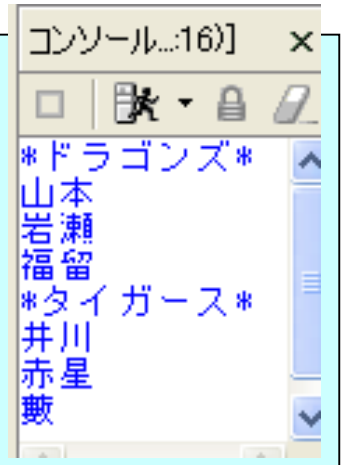

```
import java.util.ArrayList;
public class Group {
  public ArrayList<String> members = new ArrayList<String>();
  private String groupName;
  public void show(){
    System.out.println("*"+this.groupName+"*");
    for(int i=0;i<members.size();i++){
      System.out.println(members.get(i));
    }
  }
  public void setGroupName(String groupName) {
    this.groupName = groupName;
  }
}
```

（10．5．5）クラスGroupを利用するクラス

■（方法2．2）の場合のみ、メンバの登録の仕方が違います。




```
public class GroupMng {
  public static void main(String[] args) {
    String member[][] ={{"山本","岩瀬","福留"},
                        {"井川","赤星","藪"} ,};
    String groupName[]={"ドラゴンズ","タイガース",};
    Group groups[] = new Group[member.length];
    for(int i=0;i<member.length;i++){
      groups[i]=new Group();
      groups[i].setGroupName(groupName[i]);
      for(int j=0;j<member[i].length;j++){
        groups[i].add(member[i][j]);
        // groups[i].members.add(member[i][j]);
      }
    }
    for(int i=0;i<member.length;i++){
      groups[i].show();
    }
  }
}
```



（方法1）
（方法2．1）

（方法2．2）

（10．5．6）比較

■データ保護

- データ保護の観点からは、（方法2.1）が良いと考えられます。

	（方法2．1）	（方法1）、（方法2．2）
外部からのメンバー一覧の操作	・メソッド“add”の定義により、追加の操作のみが可能	・ArrayListクラスのすべてのメソッドによる操作が可能

■簡便さ、理解のしやすさ

- （方法2．1）では必要なメソッドをいちいち追加しなければなりませんが、（方法1）はその必要はありません。
- （方法2．2）ではフィールド名（例では”members”）を外部でも意識しておかなければなりませんが、（方法1）はその必要はありません。

■（方法1）と（方法2．1）とを、場合によって使い分けることになります。

（10．5．7）課題

■課題10－5－1

（1）スライド(10.5.2)(10.5.5)のプログラムを作成して動作を確認してください。

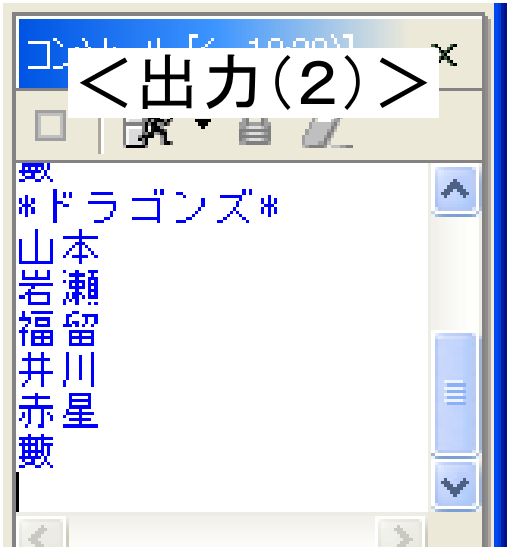
（2）上記プログラムの最後に、次の処理を追加してください（クラスGroupは変更せず、addAllメソッドを使ってください）。

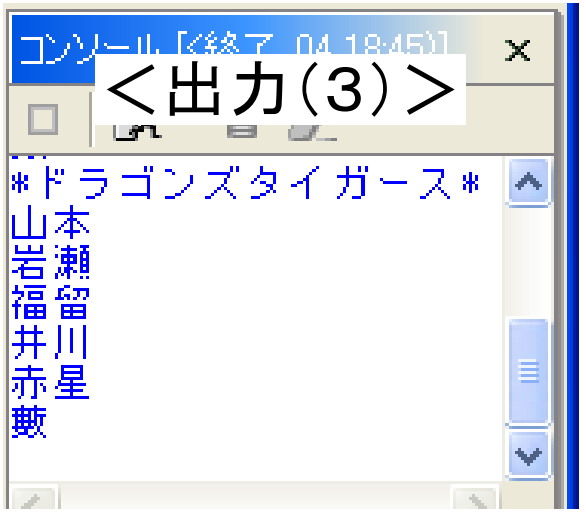
◆ドラゴンズ側に、タイガース側のメンバも追加する（阪神タイガース側のメンバはそのままでOK）。

◆ドラゴンズ側のグループを表示する。

（3）クラスGroupに次のようなメソッドを追加して、グループ名も文字列として連結されるようにしてください。

◆public void merge(Group grl);





（10．5．8）課題

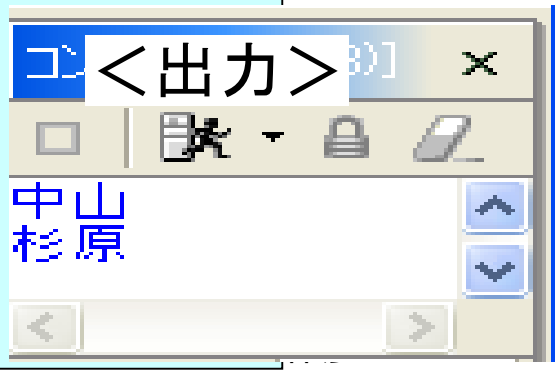
■課題10－5－2

- HashMapを継承して、次のようなクラス“HomeTown”を作成してください。
  - ◆右のような表を管理する。
    文字列として県名を引数とし、
    該県に体操する名前の一覧を
    出力するメソッド”showPeople”を持つ。

（名前）	（県名）
井戸	岐阜
杉原	愛知
中山	愛知

- 上記HomeTownクラスを利用する次のようなプログラムを作成して動かしてください。

```
public class Kadai1052 {
  public static void main(String[] args) {
    HomeTown ht = new HomeTown();
    ht.put("井戸","岐阜");
    ht.put("杉原","愛知");
    ht.put("中山","愛知");
    ht.showPeople("愛知");
  }
}
```



（10．6）アプレット（Applet）

■ここでは、スライド(10.4.2)に示したとおり、次のような例としてアプレットを扱います。従って、アプレット自体の解説にはなっていません。

- プログラムの部品化の仕組みの中で、継承して使うことが前提とされたクラスを利用する。

■アプレットとは、IEやネットスケープなどのパソコンのブラウザ上で動くJavaのプログラムです。

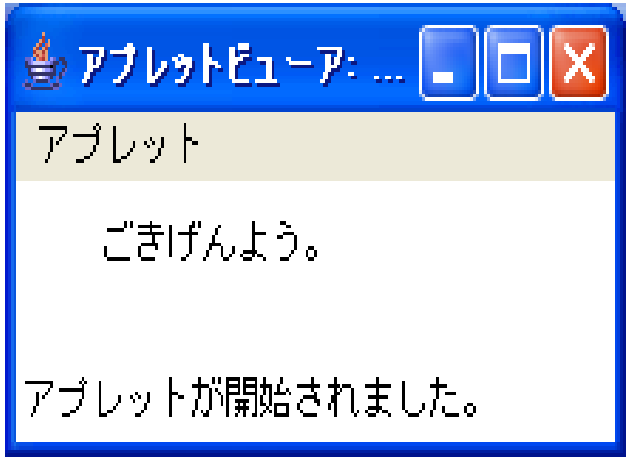
- 今はFlashにその座を奪われていますが、当初Javaが発表された時には、フロントエンド（ブラウザ側）で動作し、高速な描画等が可能なアプレットに注目が集まっていました。

■ここでは、ブラウザの替わりに“アプレット・ビューア”を用います（eclipseではこのほうが好都合です）。

（10．6．1）簡単なアプレット

■右のようなウインドウを表示する、極めて単純なアプレットを考えます。

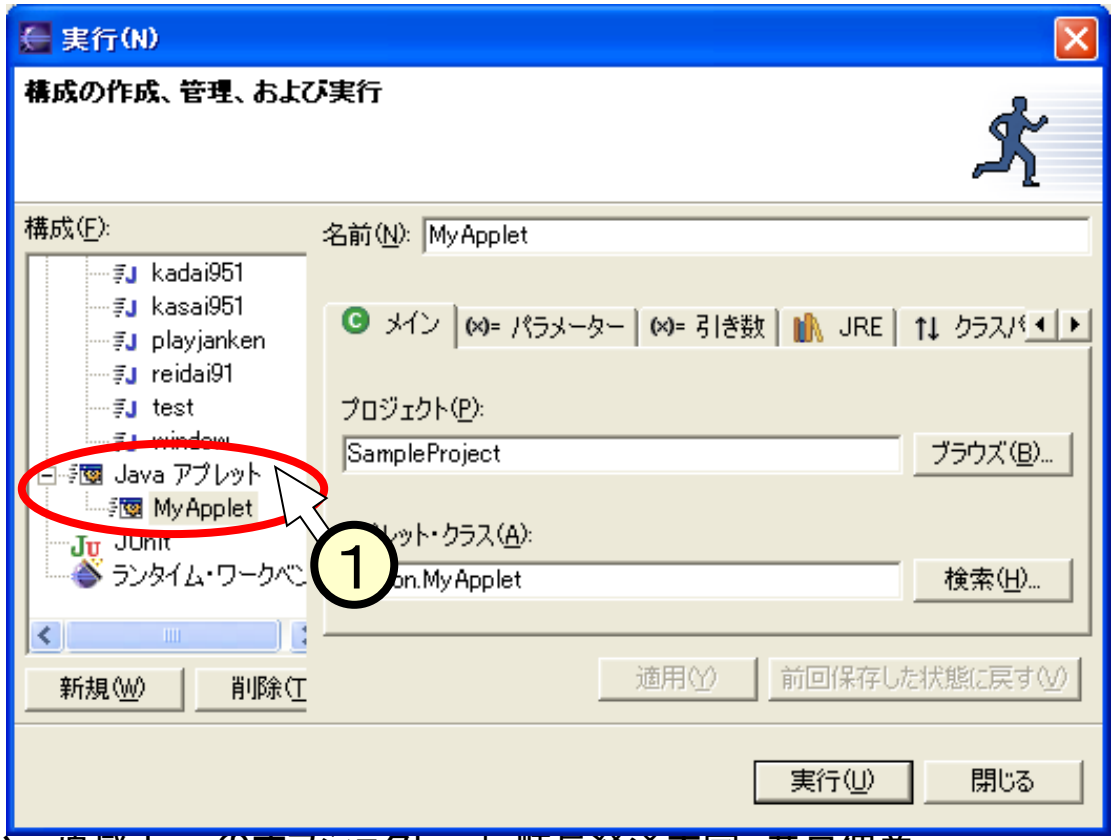
■ソースは次のとおりです。見ての通り、java.applet.Appletクラスを継承しています。メソッド”paint”は、オーバーライドを行っています。




```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet {
  public void paint(Graphics g){
    g.drawString("ごきげんよう。",30,20);
  }
}
```

（10．6．2）eclipseでのアプレットの実行

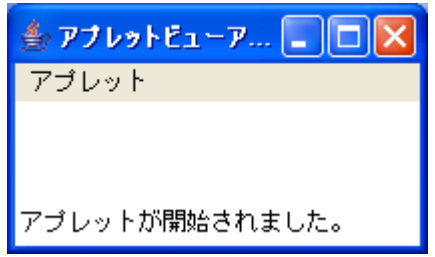
■通常のJavaアプリケーションと同様に、実行することができます。すなわち、メニューから、[実行]−[実行]をクリックし、「実行ウインドウ」中で、「構成」欄でJavaアプレットを選ぶ（①）こと以外は、Javaアプリケーションと同じです。



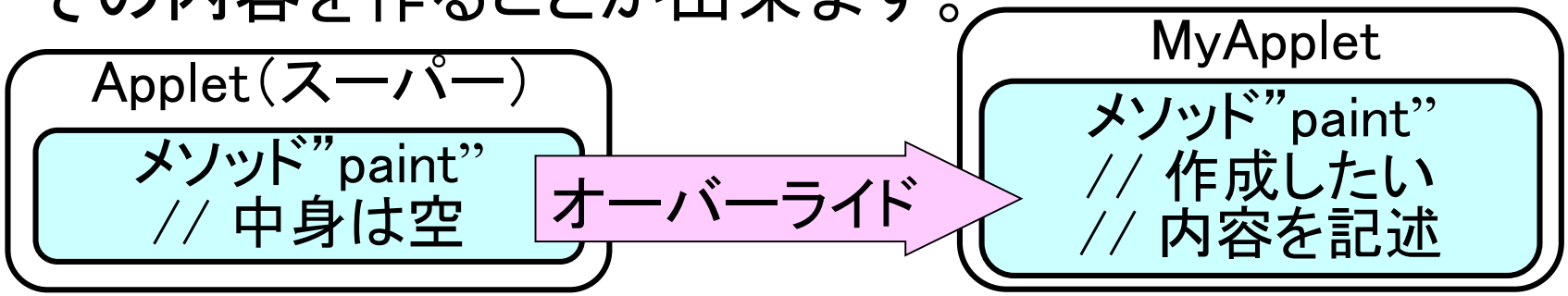
（10．6．3）動作のしくみ

■次のように、内容が空のアプレットを起動すると、何も中身は無いのですが、一応画面だけは起動されます。

```
import java.applet.Applet;
public class MyApplet extends Applet {
}
```



■上記の例では、メソッドpaintは中身のない状態です。スライド(10.6.1)のようにオーバーライドすることにより、その内容を作ることが出来ます。

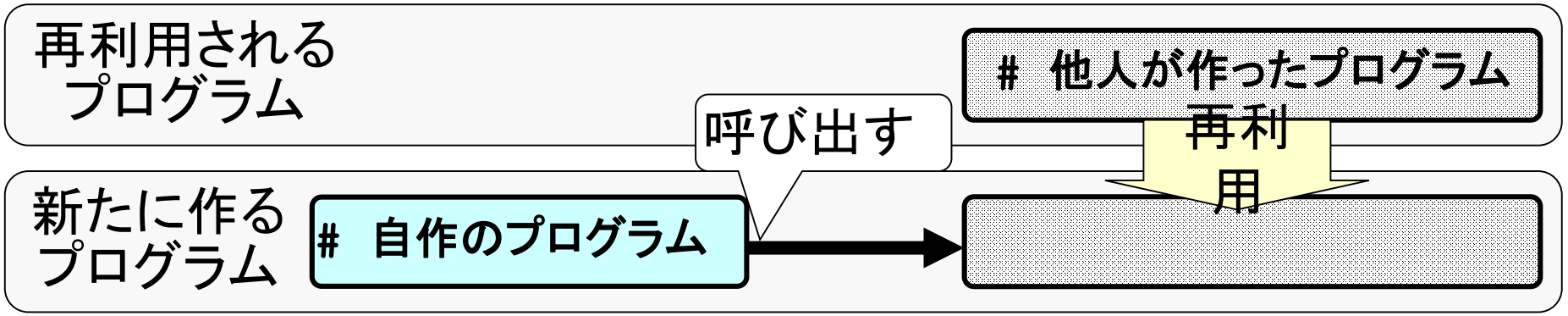


■アプレットは、このようにJava.applet.Appletクラスを継承して作る仕組みになっているわけです。

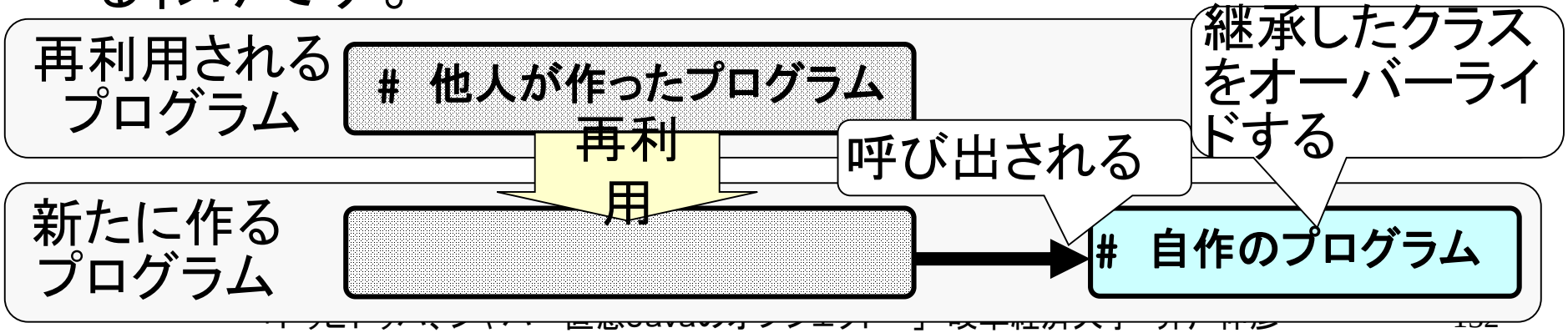
（10．6．4）プログラムの再利用の方法

■プログラムを再利用する場合、2つの方法があります。

■再利用されるプログラムを呼び出す場合は、簡単です。



■再利用されるプログラムから呼び出してもらう場合は、継承したクラスをオーバーライドすることにより実現するわけです。



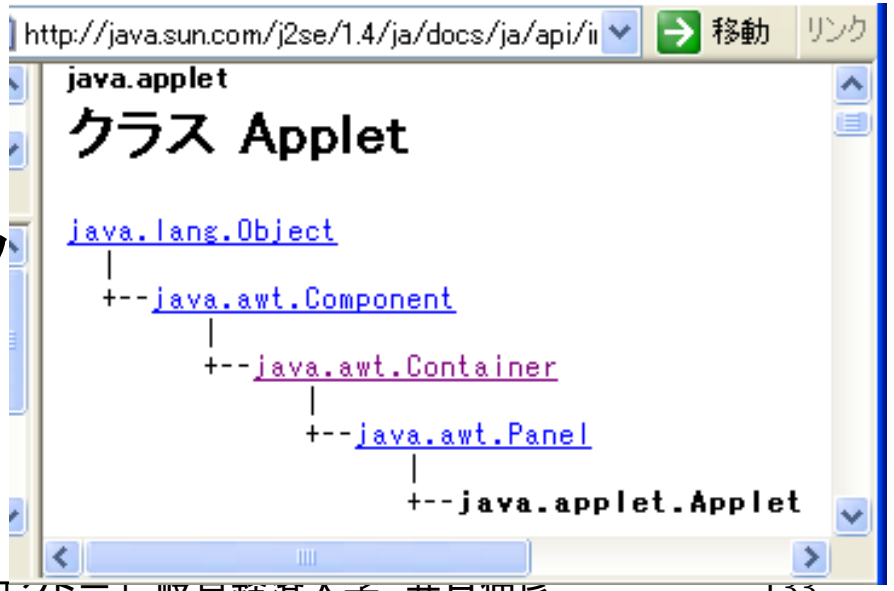
（10．6．5）paintメソッドはどこにある？

■オンラインマニュアルでAppletクラスを見ても、paintメソッドは見当たりません。その代わり次のような記述が見つかります。

クラス java.awt.Container から継承したメソッド
add, add, …….   ……, minimumSize, paint, paintComponents,….

■Appletクラスも、Contanerクラスを継承することで、paintメソッドを持っている訳です。

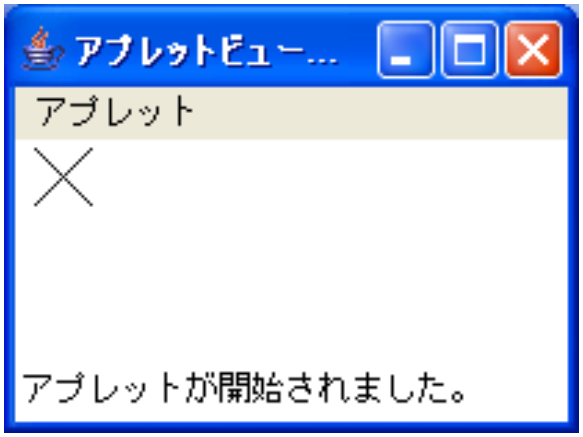
■オンラインドキュメントのクラスの説明の先頭には、該クラスの継承関係が示してあります。



（10．6．6）課題

■課題10－6－1

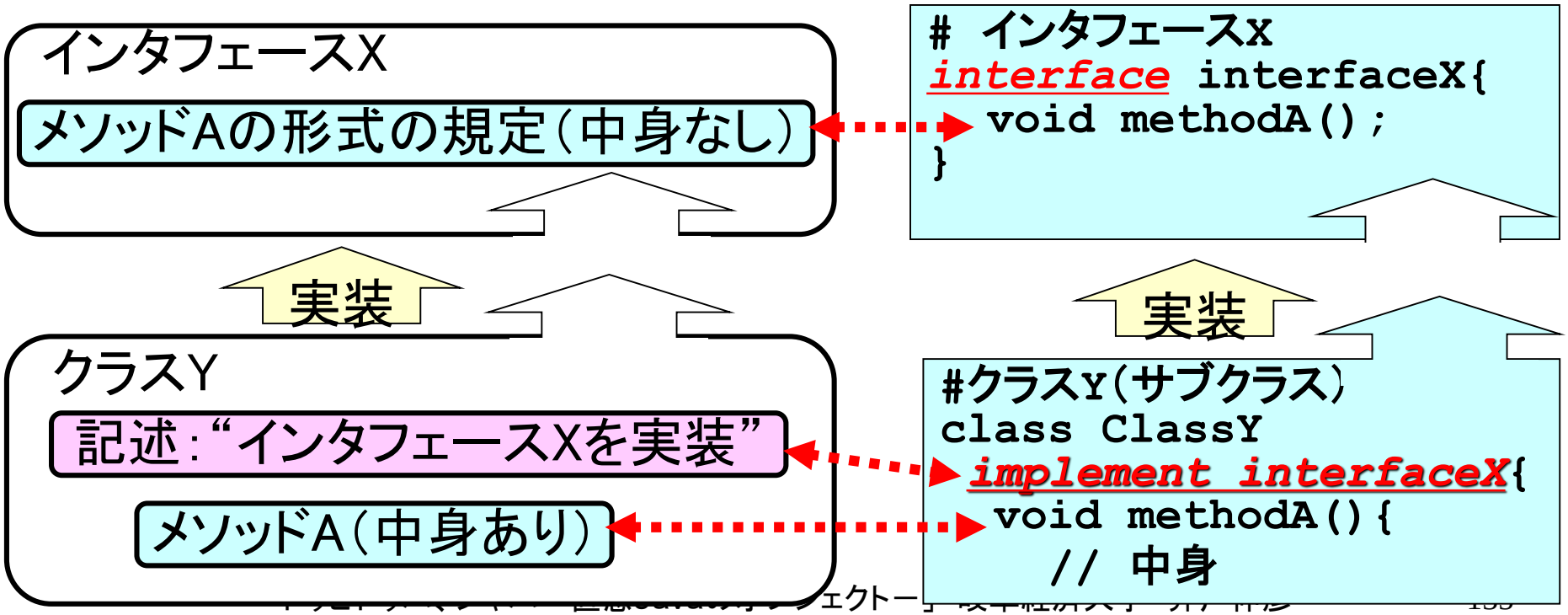
- オンラインドキュメントにて“**java.awt.Graphics**”を参照し、アプレット上に×印を描いてみてください。



（11）インタフェース

■インタフェースとは、メソッドの形式のみを規定（メソッドの中身を含まない）する、クラスに似たプログラムの単位です。

■インタフェースは、これを実装（インプリメント：implement）するクラスで、インタフェースで規定されたメソッドの中身を記述するような方法で利用されます。



（11．1）例：選手の年棒

■インタフェース“Income”をインプリメントしているクラス“BPlayer4”では、“void showIncome()”というメソッドを作成しないと、コンパイル・エラーになります。

<Income.java(sports)>

```
package sports;
public interface Income {
        void showIncome();
}
```

②“Income”で定義されたメソッド“showIncome”を、

<BPlayer4.java(baseball)>

```
package baseball;
import sports.Income;
public class BPlayer4 implements Income{
(中略)
  public void showIncome() {
    System.out.println(name+"：年棒："+runs*500+"万円");
  }
}
```

①このクラスは“Income”を実装しているので、

③メソッドとして持たないとコンパイルエラーとなる

（11．2）同じメソッドを持つことを強制する

- インタフェースを用いると、これを実装するクラスに規定どおりのメソッドを持つことを強制出来るわけです。
- 例えば、クラス“**Player**”でもインタフェース“**Income**”を実装することとすれば、“**BPlayer4**”と同じメソッドで呼び出すことが保証されます。

<BPlayer.java(baseball)>

```
public class BPlayer implements Income{
        :
  public void showIncome() {
        :
```

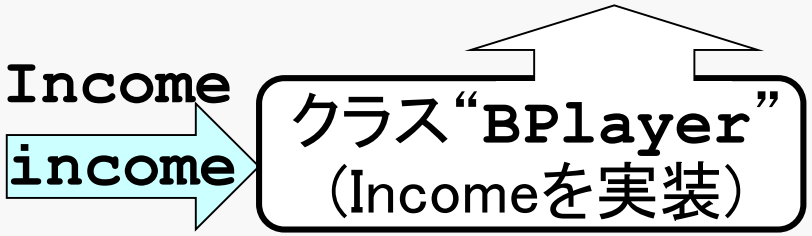
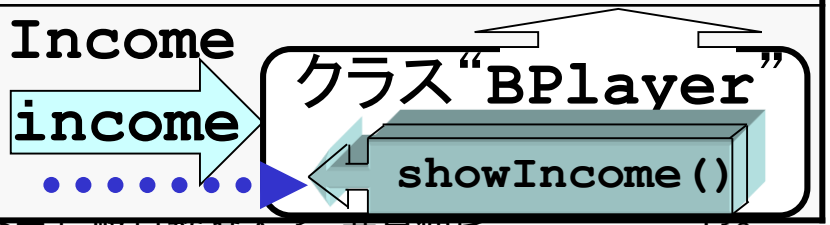
<Player.java(football)>

```
package football;
import sports.Income;
public class Player4 implements Income{
(中略)
  public void showIncome() {
    System.out.println(name+":年棒:"+goals*1000+"万円");
  }
}
```

（11．3）インタフェース型の変数

■クラス型の変数（スライド(6)参照）と同様に、インタフェース型の変数を定義することができます。

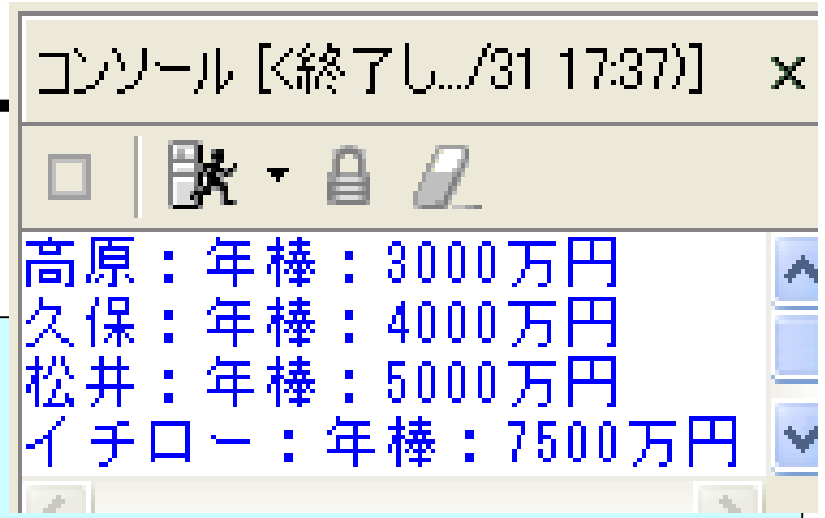
■インタフェース型の変数には、そのインタフェースを実装したクラスのインスタンスを保持することができます。

操作	記述	操作後の状態
インタフェース型の変数を用意する	`Income income;`	
該インタフェースを実装するクラスのインスタンスを関連付ける	`income` `=new Player();`	
インタフェースで規定されたメソッドを呼び出す	`income.showIncome();`	

（11．4）利用例

<ProfessionalSports.java(sports)>

```
 1:package sports;
 2:import baseball.BPlayer4
 3:import football.Player4;
 4:public class ProfessionalSports {
 5:   public static void main(String[] args) {
 6:      Income incomes[] = new Income[4];
 7:      incomes[0] = new Player4("高原",3);
 8:      incomes[1] = new Player4("久保",4);
 9:      incomes[2] = new BPlayer4("松井",100);
10:      incomes[3] = new BPlayer4("イチロー",150);
11:      for(int i=0;i<incomes.length;i++){
12:        incomes[i].showIncome();
13:      }
   :   }
   :}
```

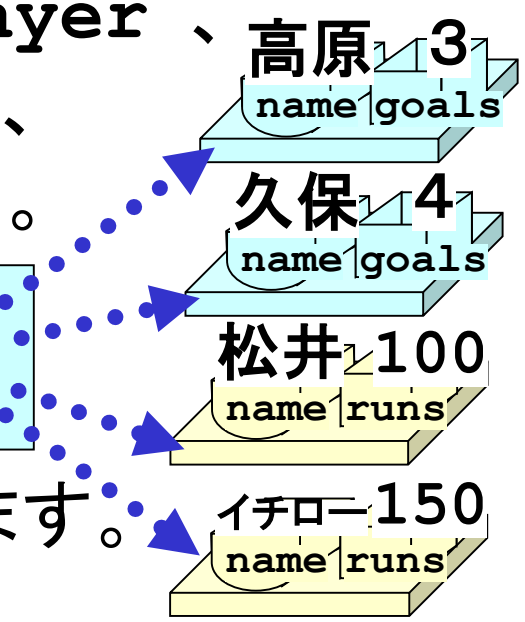


このコンストラクタは
追加してください。

（11．4．1）より一般的な手順

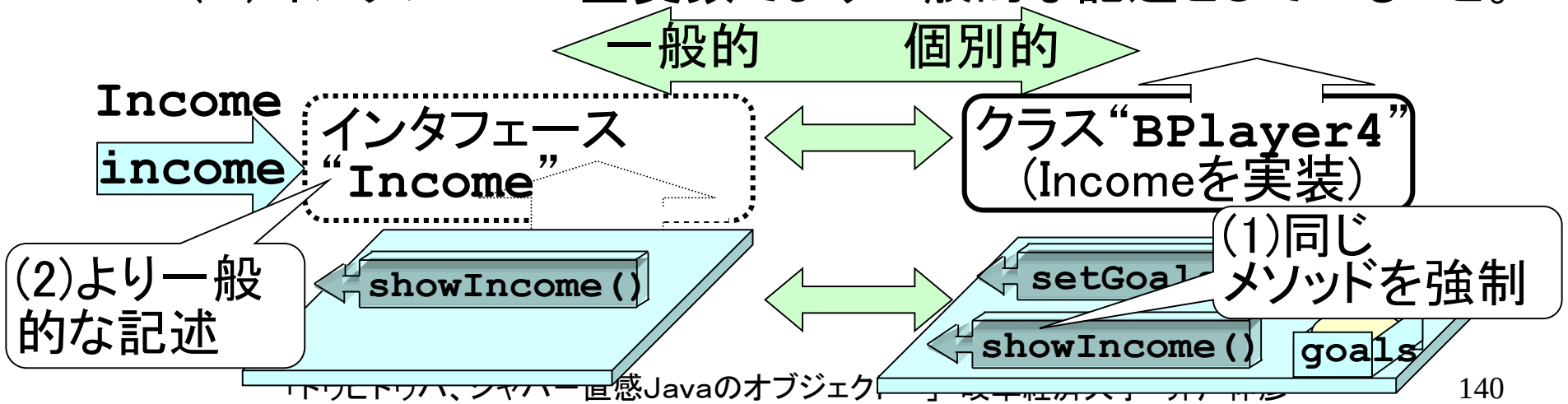
■スライド(11.4)のプログラムでは、“BPlayer”、“Player”の両方のインスタンスに対して、同じループでメソッドを呼び出しています。

```
11: for(int i=0;i<incomes.length;i++){
12:         incomes[i].showIncome();
13: }
```



■これは、次の２点により可能となっています。

- （１）同じメソッドを強制していること。
- （２）インタフェース型変数でより一般的な記述としていること。



（11．4．2）型変換

■インタフェース型の変数に代入したインスタンスは、そのままではインタフェースにないメソッドを呼び出すことは出来ません。

■インタフェースにないメソッドを呼び出す場合は、次のような型変換を行います。

<ProfessionalSports.java(sports)に追加>

```
14:     //incomes[0].show();
15:     ((Player)incomes[0]).show();
```

これは出来ない

①クラス“Player”に型変換をして、

②インタフェースにないメソッドを呼び出す

（11.5）どのような場合にインタフェースを使うか?

■継承と同様に、インタフェースもモデル化がうまくいっている場合に使用出来るわけです。よって、自作のインタフェースを実装した、自作のクラスを作成する機会は、必ずしも多いということではありません。

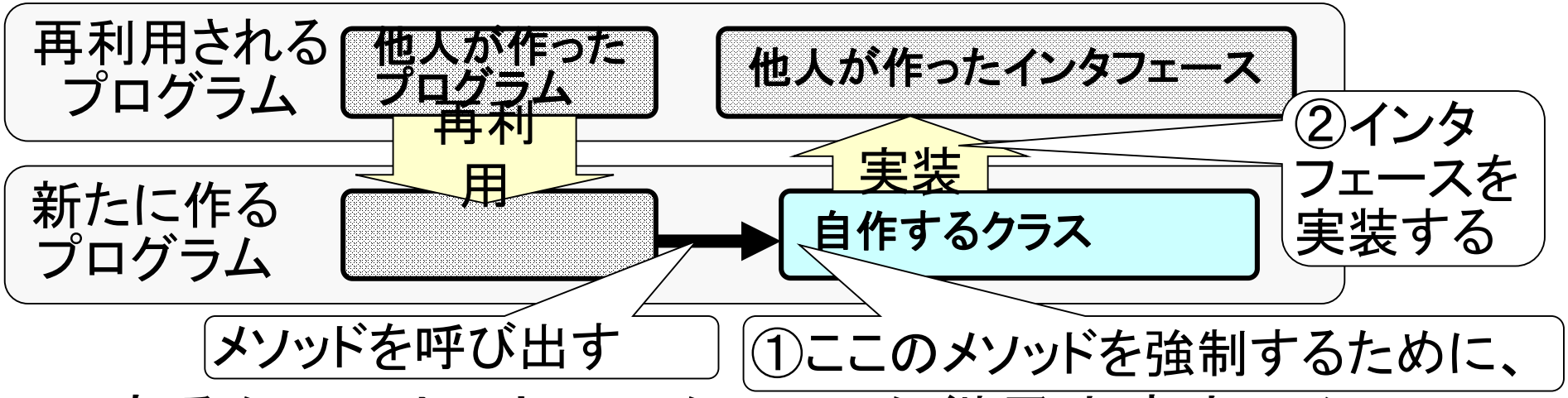
■しかしながら、継承と同様に、既存のインタフェースを用いることは良く行われます。次の2つの例を見ていきます。

（1）既存のインタフェースにより、メソッドの実装を強制する（スライド(11.6)）。

（2）インタフェース型の変数を用いることで、より一般的な処理とする（スライド(11.7)）。

（11．6）メソッドを強制する機能

■この機能は、スライド(10.6)で見た、継承によるプログラム際利用の例と似ています。すなわち、次のように既存のプログラムとインタフェースを利用します。

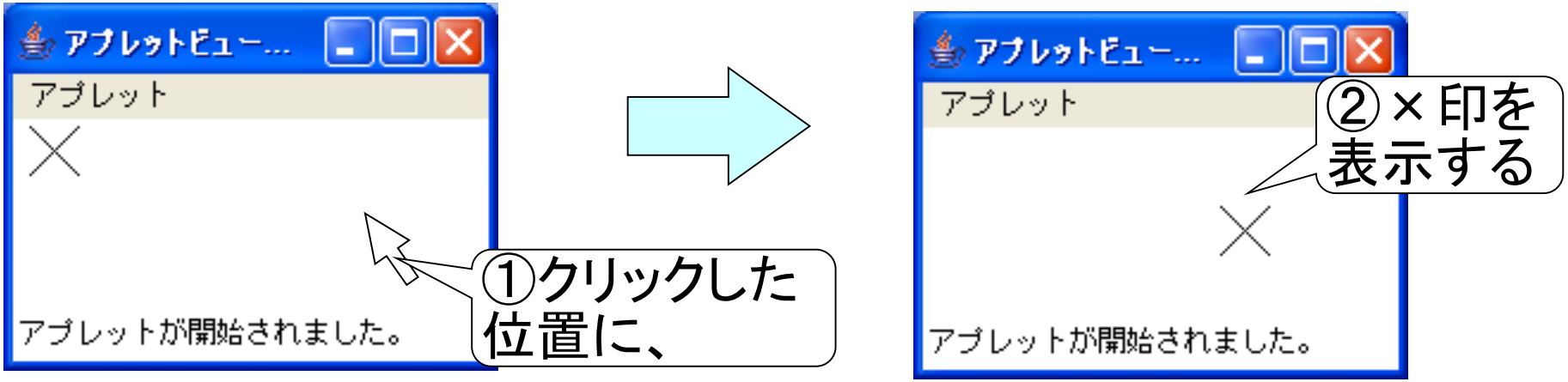


■あるクラスはひとつのクラスしか継承出来ませんでしたが、インタフェースを複数実装することはできます。よって、様々なメソッドを既存のプログラムから呼び出されるようなクラスを作成する場合には、インタフェースを用いることになります。

（11．6．1）マウスのクリック

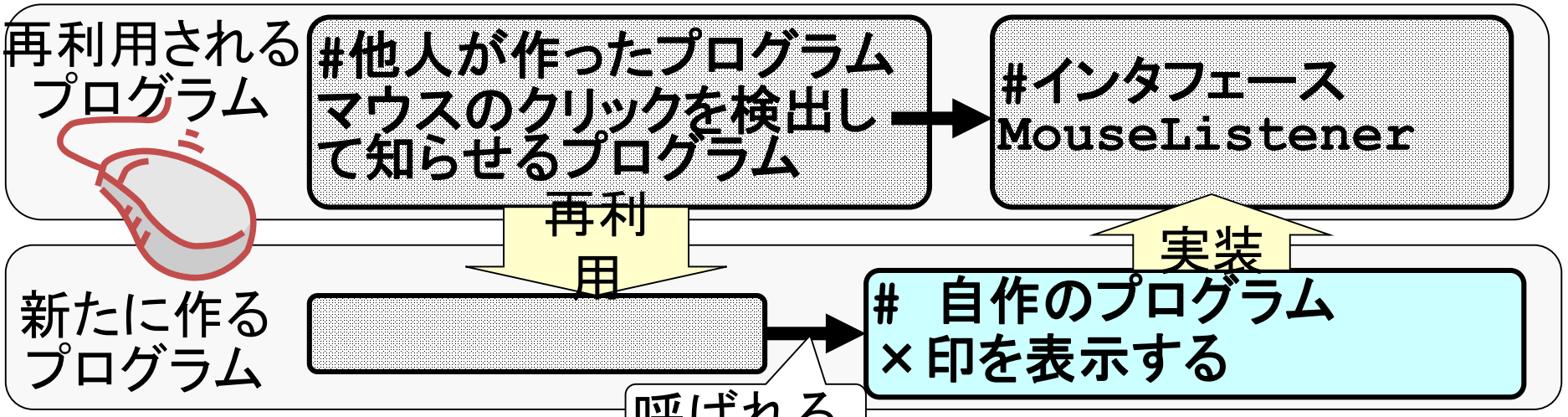
■自作部分が呼ばれるような場合はたくさんあります。

■例えば、マウスがクリックされたら、そこに×印を表示するアプレットを作ることにします。



（11．6．2）インタフェースの実装

- 「マウスがクリックを検出して知らせるプログラム」の部分は、一般的な機能ですので、既存プログラムを再利用できます。
- 新たに作るプログラムの中では、「×印を表示する機能」を作れば良いのですが、これは再利用した部分からメソッドが呼ばれることになります。
- このメソッドの形式を強制するために、インタフェース“MouseListener”を実装します。



（11．6．3）ソース

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
public class MouseClick extends Applet
                        implements MouseListener{
  int x=10,y=10;
  public void init(){
    addMouseListener(this);
  }
  public void paint(Graphics g){
    g.drawLine(x-10,y-10,x+10,y+10);
    g.drawLine(x-10,y+10,x+10,y-10);
  }
  public void mouseClicked(MouseEvent me) {
    x=me.getX();
    y=me.getY();
    repaint();
  }
  public void mouseEntered(MouseEvent me) {}
  public void mouseExited(MouseEvent me) {}
  public void mousePressed(MouseEvent me) {}
  public void mouseReleased(MouseEvent me) {}
}
```

×印を描く処理

これらのメソッドを作成することが必須となる

これによりpaintが呼ばれる

（11．6．4）課題

■課題11－6－1

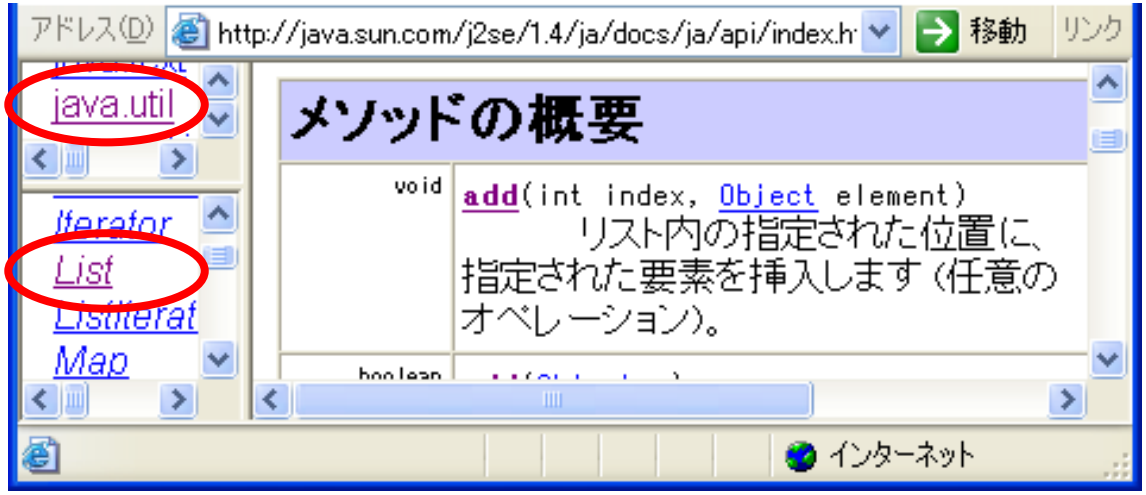
- マウスボタンを離すと○印を描くように、スライド(11.6.3)のアプレットを変更してください。

（11．7）より一般的な変数

■標準クラスの“**ArrayList**”（スライド(9.4)）は、インタフェース“List”を実装しています。このことは、もちろんオンラインドキュメントにも載っています（①）。

（11．7．1）インタフェースList

■インタフェース“List”を見てみると、スライド（9.4）で説明したようなメソッドは、すべて含まれています。

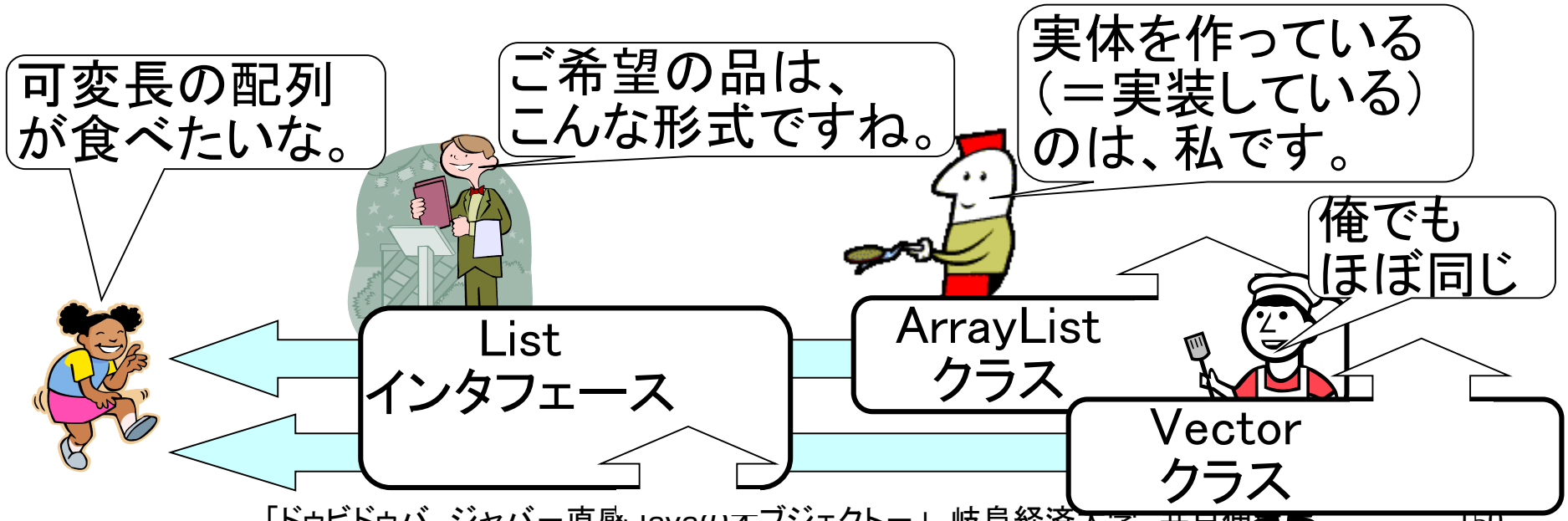


■実際のところ、ArrayListのドキュメントの先頭近くには、次のような記述があります。

**List インタフェースのサイズ変更可能な配列の実装です。
リストの任意のオペレーションをすべて実装し、null を含むすべての要素を許容します。**

（11．7．2）ListとArrayListクラス、Vectorクラス

- ArrayListクラスはListインタフェースに記されたメソッドを実現するためのクラスです。
- Listインタフェースを実装するクラスには、このほかVectorクラスがあります。
- ArrayListとVectorとは、Listインタフェースの規定内では、ほぼ同様に使用することが出来ます（実際は同期化の点が異なりますが、これについては触れません）。



(11.7.3)例

<実行結果>

```
# Strings longer than 6
sekigahara
# Strings longer than 5
sekigahara
hikone
nagoya
```

```
 1:import java.util.ArrayList;
 2:import java.util.List;
 3:import java.util.Vector;
 4:public class ListTest {
 5:   public static void main(String[] args) {
 6:     String placeNames[]
 7:     ={"ogaki","gifu","sekigahara","hikone","nagoya",};
 8:     ArrayList<String>arl = new ArrayList <String>();
 9:     Vector <String> vct = new Vector <String>();
10:     for(int i=0;i<placeNames.length;i++){
11:       arl.add(placeNames[i]);
12:       vct.add(placeNames[i]);
13:     }
14:     showLonger(arl,6);
15:     showLonger(vct,5);
16:   }
17:   private static void showLonger(List<String>lst
                                      ,int strLen){
18:     System.out.println("#Strings longer than"+strLen);
19:     for(int i=0;i<lst.size();i++){
20:       String tempStr=(String)lst.get(i);
21:       if(tempStr.length()>strLen){
22:         System.out.println(tempStr);
23:       }
24:     }
25:   }
26:}
```

（11．7．4）Listの入力パラメータ

■前スライドのプログラムでは、showLongerメソッドの入力パラメータをListとすることにより、ArrayListクラス、Vectorクラスの両方のインスタンスを受け付けることが出来るようになっています。

```
 8: ArrayList arl = new ArrayList();
14: showLonger(arl,6);
9: Vector vct = new Vector();
15: showLonger(vct,5);
17: private static void
        showLonger(List lst,int strLen){
```

<String>を省略しています

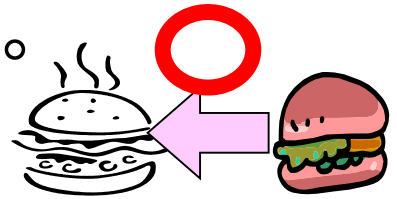
■このように、入力パラメータをより一般的な変数（すなわちList）としたほうが、メソッドが使える場合が増えて、利便性が増すことになります。

（11．7．5）代入が可能であること

■インタフェース型の変数に、そのインタフェースを実装するクラスのインスタンスは代入が可能です。

```
ArrayList arl = new ArrayList();
List lst = arl;
```

OK



- ArrayListは、Listと見なすが出来る。

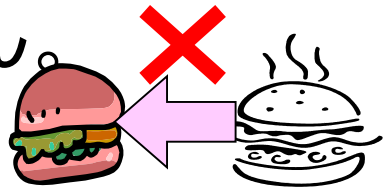
（マクドのハンバーガーは、ハンバーガーと見なすことが出来る）

■クラス型の変数に、そのクラスが実装するインタフェース型の変数の値を代入することは出来ません。

```
List lst = new ArrayList();
ArrayList arl = lst;
```

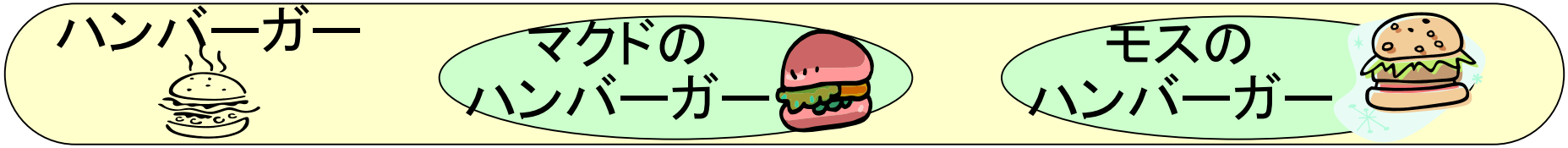
NG

コンパイルエラー



- Listは、ArrayListとは限らない。

（ハンバーガーは、マクドのハンバーガーとは限らない）



（11．7．6）どちらの変数とすべきか？

■次の2つの書き方は、いずれも用います。

```
(A):ArrayList arl = new ArrayList();
```

```
(B):List lst = new ArrayList();
```

- 両者の違いは、(A)マクドのハンバーガーにこだわるか、(B)ハンバーガーなら何でもよいかの違いとなります。
- 一般的には、(B)のほうが合理的な場合が多そうですが、ArrayListとVectorとの違いも説明してませんので、ここでは議論しません。

■HashMapクラスはMapインタフェースの実装ですので、次のような書き方を良く目にします。

```
Map map = new HashMap();
```

- 少なくとも、このような書き方に驚かないようにしておいてください。

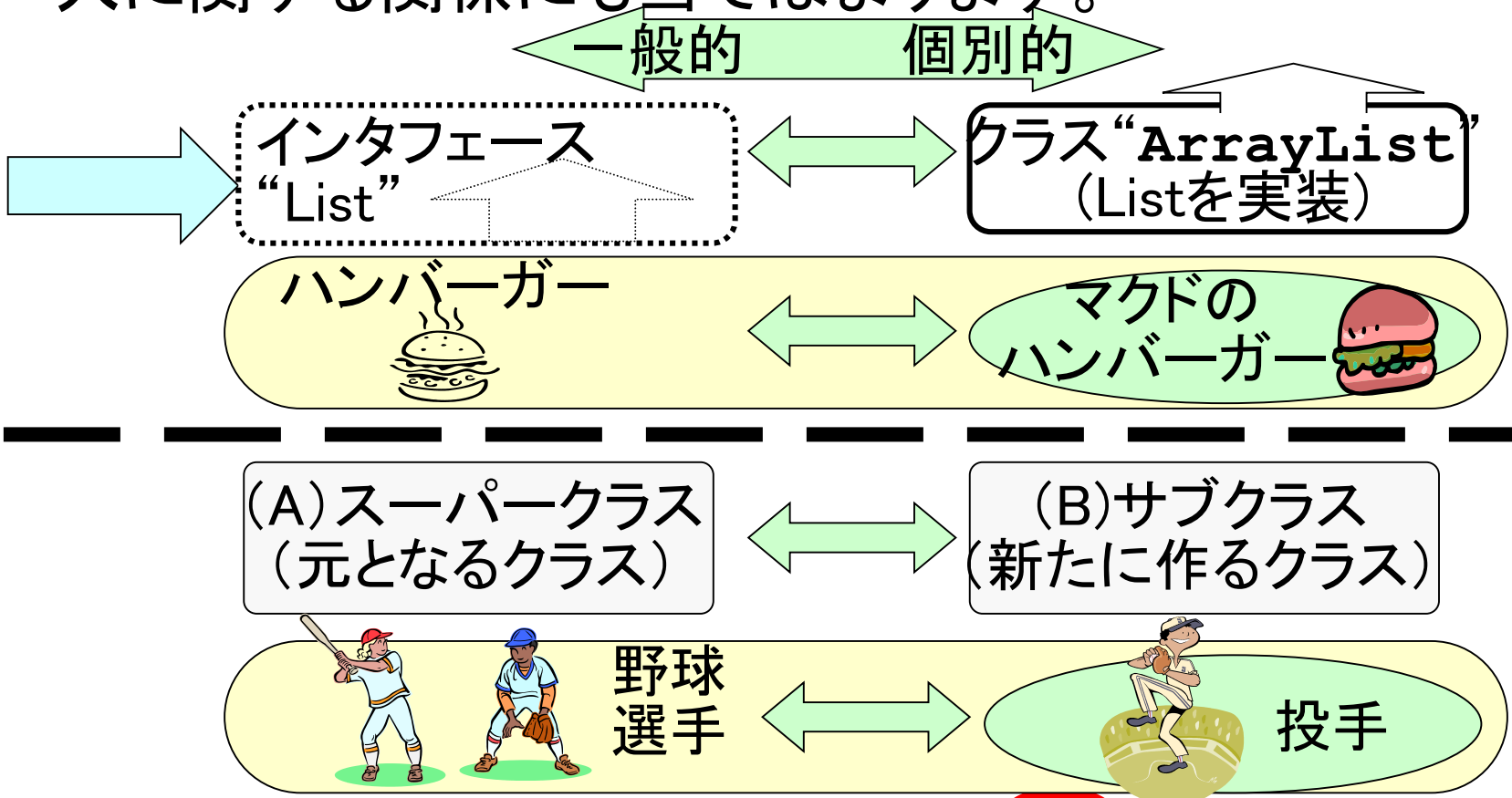
（11．7．7）どのクラスのインスタンスか？

■例えば、Listインタフェース型の変数に、ArrayListクラスのインスタンスを設定した場合、
そのことは“instanceof”演算子により判定することができます。


```
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
public class Ex1177 {
   public static void main(String[] args) {
      List<String>list0 = new ArrayList<String>();
      List<String>list1 = new Vector<String>();
     if(list0 instanceof ArrayList){
        System.out.println("list0はArrayList。");
     }else if(list0 instanceof Vector){
        System.out.println("list0はVector。");
     }
     if(list1 instanceof ArrayList){
        System.out.println("list1はArrayList。");
     }else if(list1 instanceof Vector){
        System.out.println("list1はVector。");
     }
   }
}
```

（11．8）継承における、より一般的な変数

■インタフェースとそれを実装するクラスとの代入に関する関係は、継承でのスーパークラスとサブクラスとの代入に関する関係にも当てはまります。



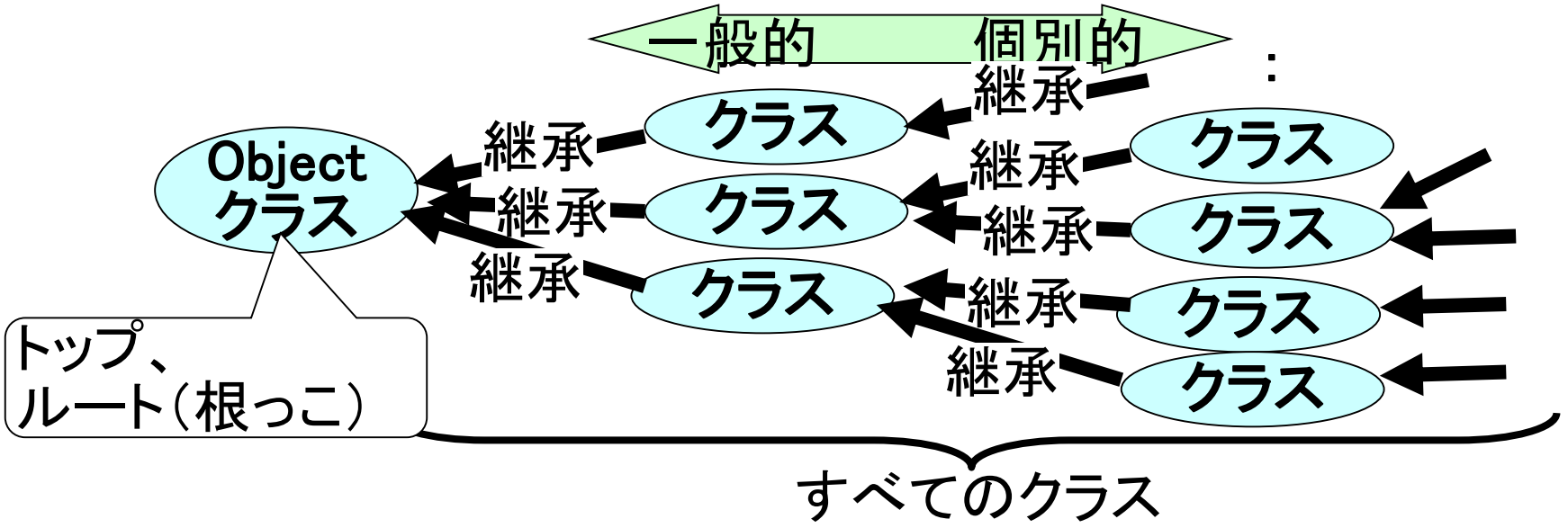

```
BPlayer bpl = new Pitcher();
```

OK

スライド
(10.2.4)参照

（11．8．1）“Object”クラス

■Objectクラス（java.lang.Object）とは、継承の階層において、全クラスのトップ（ルート）になっているクラスです。



■オンライン・ドキュメントのどのクラスを見ても、継承関係の一番上位にあるのは、Objectクラスです。

（11．8．2）自作のクラス

■みなさんが継承について記述していないクラスを自作した場合も、そのクラスObjectクラスを継承しています。すなわち、下記(A)は、(B)の省略形になっています（実際、(B)のように書いてもOKです）。

(A)
```
    :
public class FootballTeam {
    :
```

省略形

(B)
```
    :
public class FootballTeam extends Object {
    :
```

（11．8．3）Objectクラス型の変数

■以上のように、Objectクラスはすべてのクラスのスーパークラスにあたる訳ですから、オブジェクト型の変数には、すべてのインスタンスの代入が可能となります。

■ArrayListクラスのドキュメントを見ると、可変長配列に追加される要素の型は、Objectです。取り出した要素の型も、Objectです。このようにして、ArrayListはすべてのクラスを保持することが出来るようになっています（要素を取り出したときには、型変換が必要でしたね）。

メソッドの概要（ArrayListクラスのドキュメント）

追加する要素は、Object型

取り出した要素は、Object型

boolean	**add**(Object o) リストの最後に、指定された要素を追加します。
Object	**get**(int index) リスト内の指定された位置にある要素を返します。

（11．8．4）課題

■課題11－8－1

- Objectクラス型の引数を取る次のようなメソッド“showList”を、下記リストのように作成してください。
  - ◆引数がArrayListの場合は、その要素の一覧を出力。
  - ◆引数がHashMapの場合は、キーと値のペアの一覧を出力。

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;
import java.util.Map;
public class Kadai1181 {
  public static void main(String[] args) {
    ArrayList<String>arl = new ArrayList <String>();
    arl.add("小泉");arl.add("岡田");arl.add("管");
    HashMap<String,String>map = new HashMap <String,String>();
    map.put("明治","1868");map.put("大正","1912");
    showList(arl);
    showList(map);
  }
  private static void showList(Object object){
    // ここの部分のプログラムを作成する。
  }
}
```