(6) クラスとインスタンス

(6．1)
クラスと
インスタンス

インスタンスとは何か
を考えます。

(6．2)
プログラムでの
クラスとインスタンス

プログラムでのインス
タンスのイメージを考
えます。

(6．3)
UMLのクラス図

クラスの内容とクラス
の関係とを、クラス図
に描きます。

飛行機オブジェクト

(6．4)
ブラックジャックの
クラス図

クラス図とプログラム
との対応を見ます。

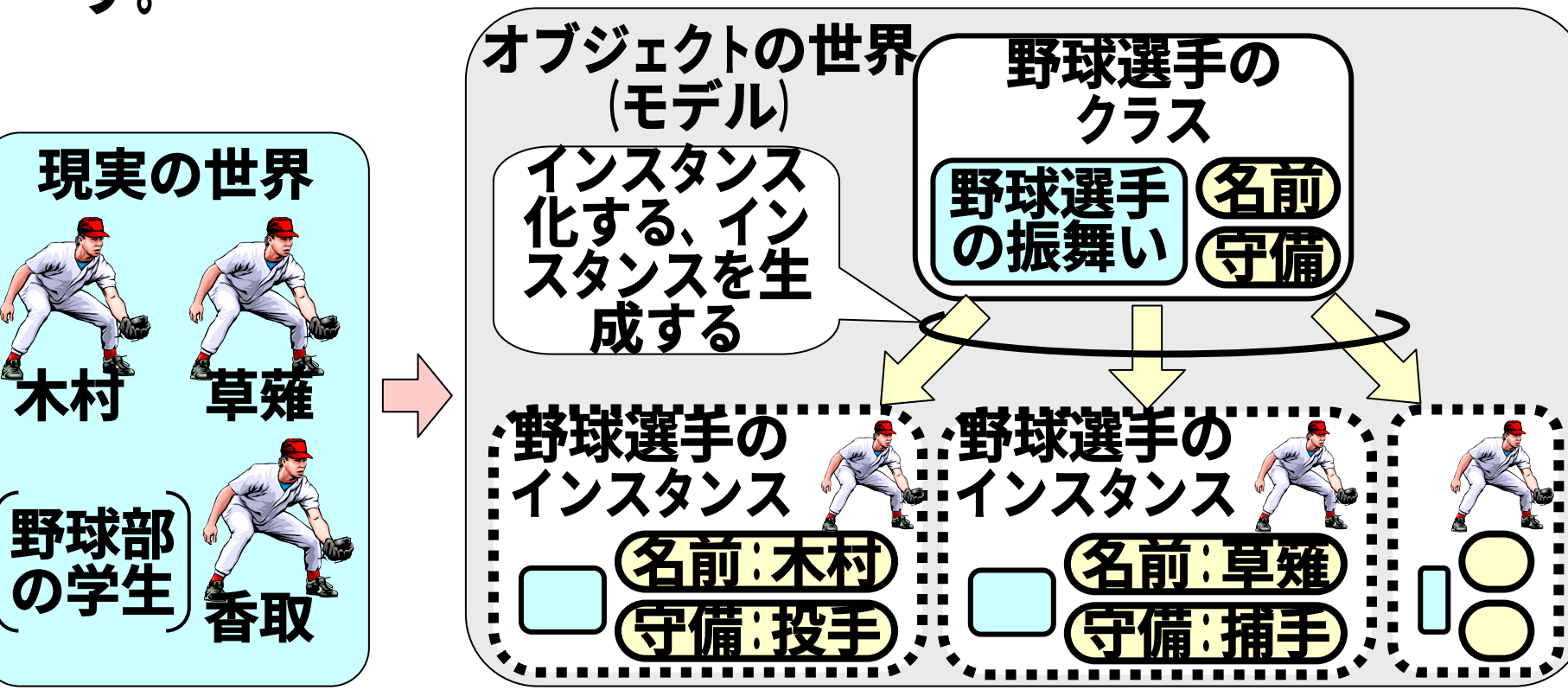
(6．1．1) 野球チームの例

■野球チームに関してモデリングを行う際、ひとりひとりの選手をオブジェクトとして捉えるとします。

■すると、各々の選手のオブジェクトは、同じ“野球選手としての振舞い”を持つことになります。プログラムについて言えば、おなじコードがいくつも必要になりますから、これは無駄です。

(6．1．2) クラスとインスタンス

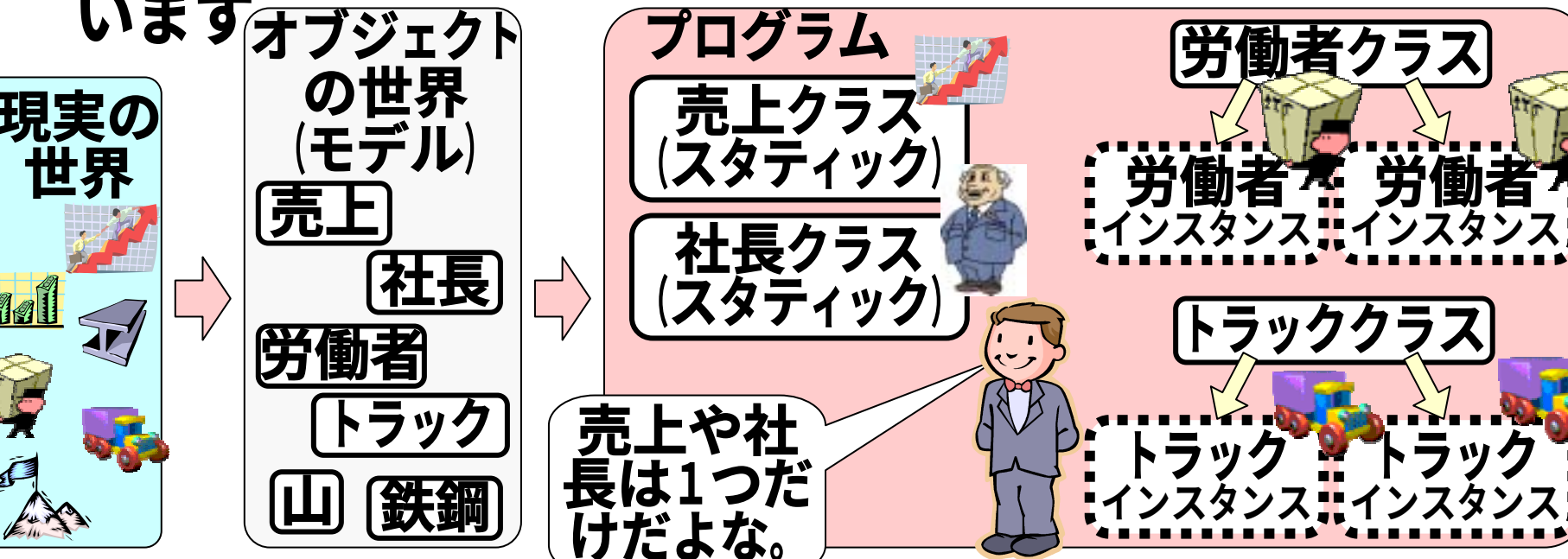
■野球チームの例で考えれば、野球選手としての定義はひとつだけ行い(これをクラスと言います)、このクラスをもとに、それぞれの属性の値を持つオブジェクト(これをインスタンスと言います)を作るほうが合理的です。



(6．1．3) スタティックなクラス

■野球選手のように、同じ属性 (データ) と振舞い (メソッド) とを持つものが複数ある場合は、クラスとインスタンスとの関係が生じます。

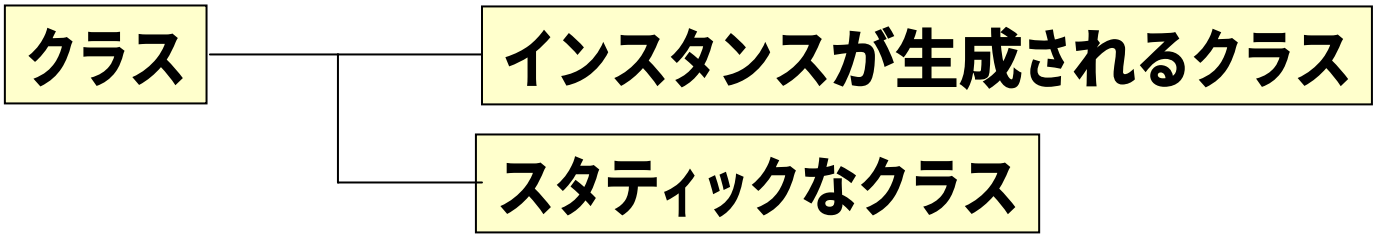
■しかしながら、考えているシステムに1つしかない“もの”は、クラスとインスタンスとの関係にする必要はありません。このようなクラスをスタティックなクラスと言います



(6．1．4) クラスに関するちいさな整理

■“もの”について、ここまで“オブジェクト”と言ってきましたが、ここからは、“クラス”と言うこととします。

■クラスは、“インスタンス”が生成される“クラス”と、発生しない“クラス” (スタティックなクラス) のいずれかということでひとまず理解してください。

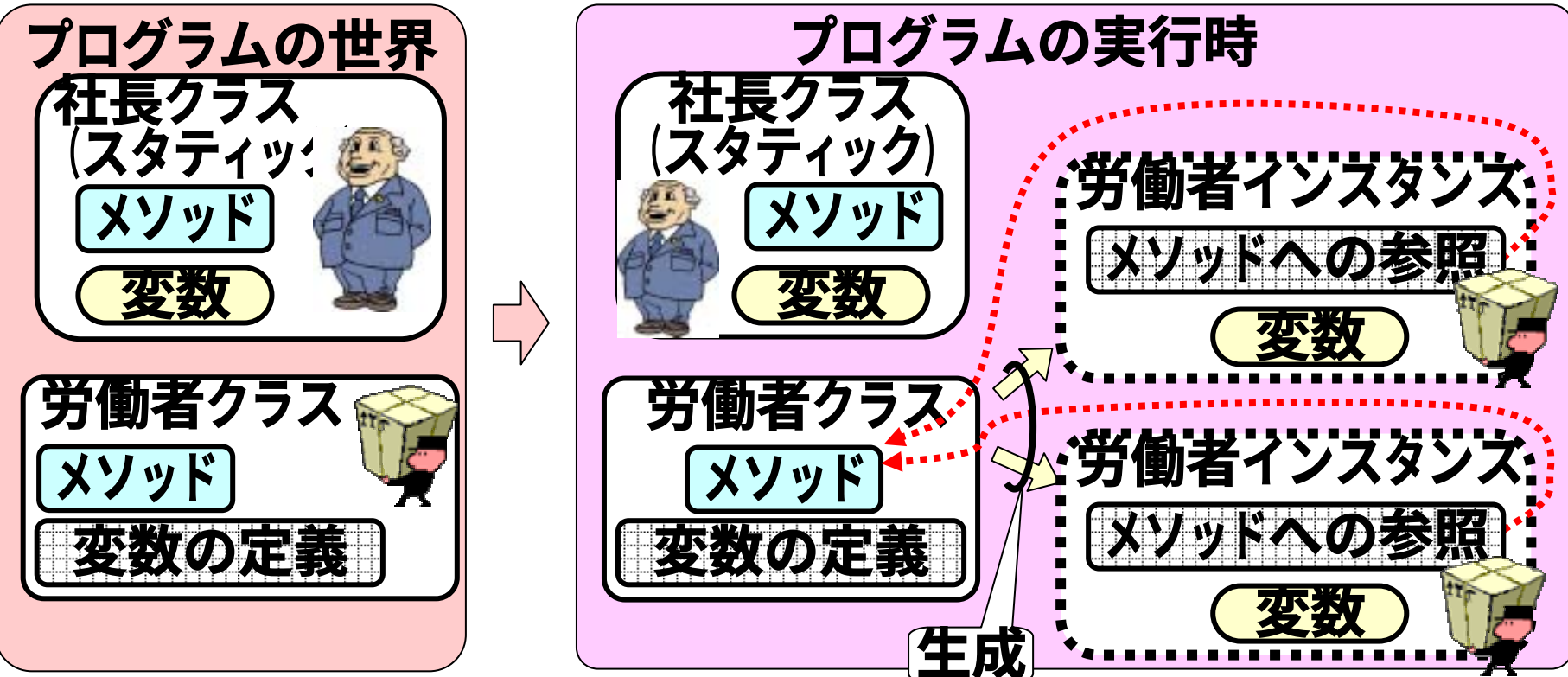


■両者の違いは、プログラムの実行時 (コンピュータ上でプログラムが起動されて動くとき) にどうなるかを見ると、明確に理解できます。

(6．2．1) プログラムの実行時

■スタティッククラスは、プログラムの時点でメソッド･変数とも既に用意されていて、実行時もそのままです。

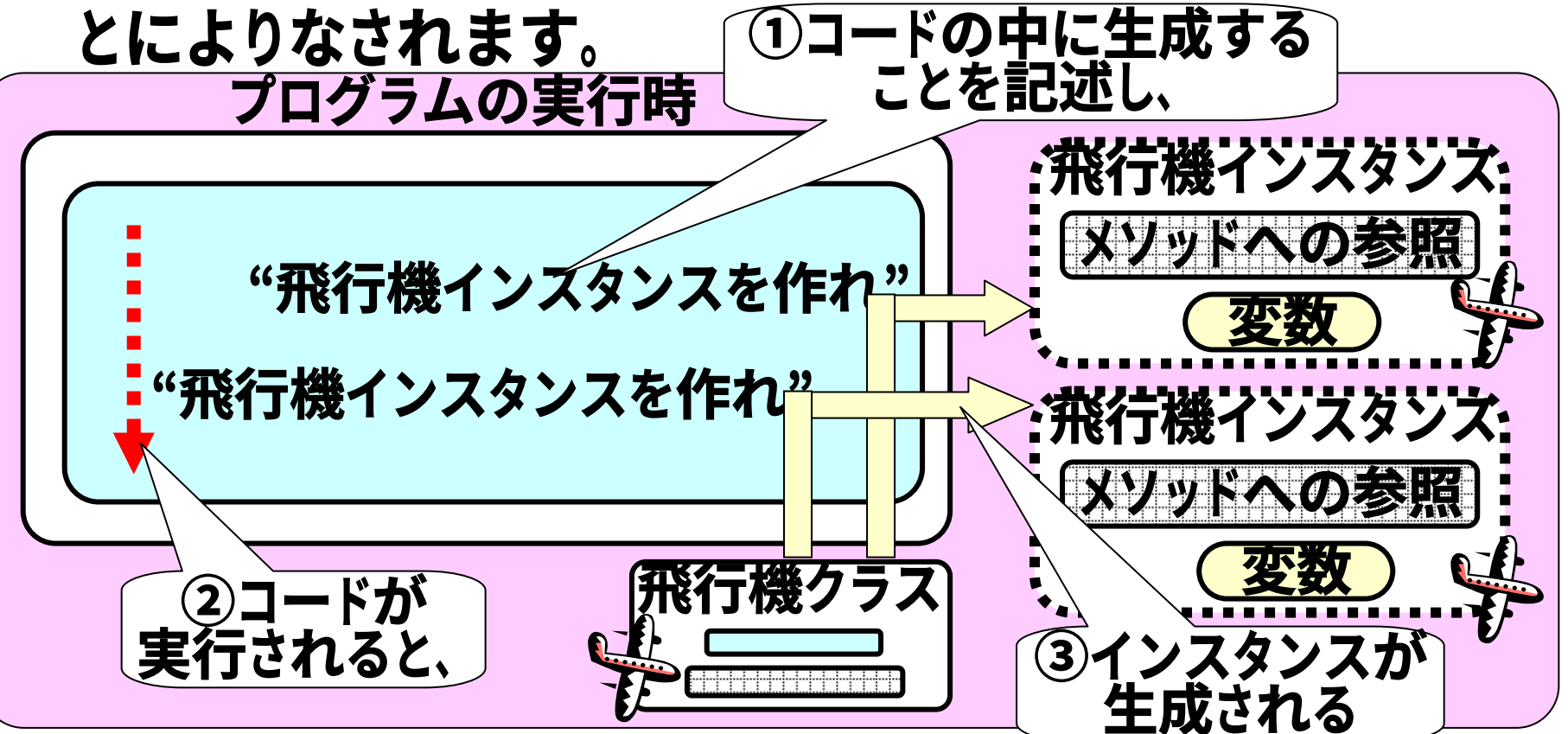
■インスタンスを持つクラスは、プログラムの時点では変数は定義しかなく、実行時のインスタンス生成により初めて生成されます。メソッドは生成されず、クラスにある実体を参照します。



(6.2.2) 誰がインスタンスを生成するのか？

■スタティッククラスは、プログラムに書いてありさえすれば、実行時にその実体が存在することになります。

■インスタンスの生成は、プログラムのコードの中に生成することを記述しておき、そのコードが実行されることによりなされます。

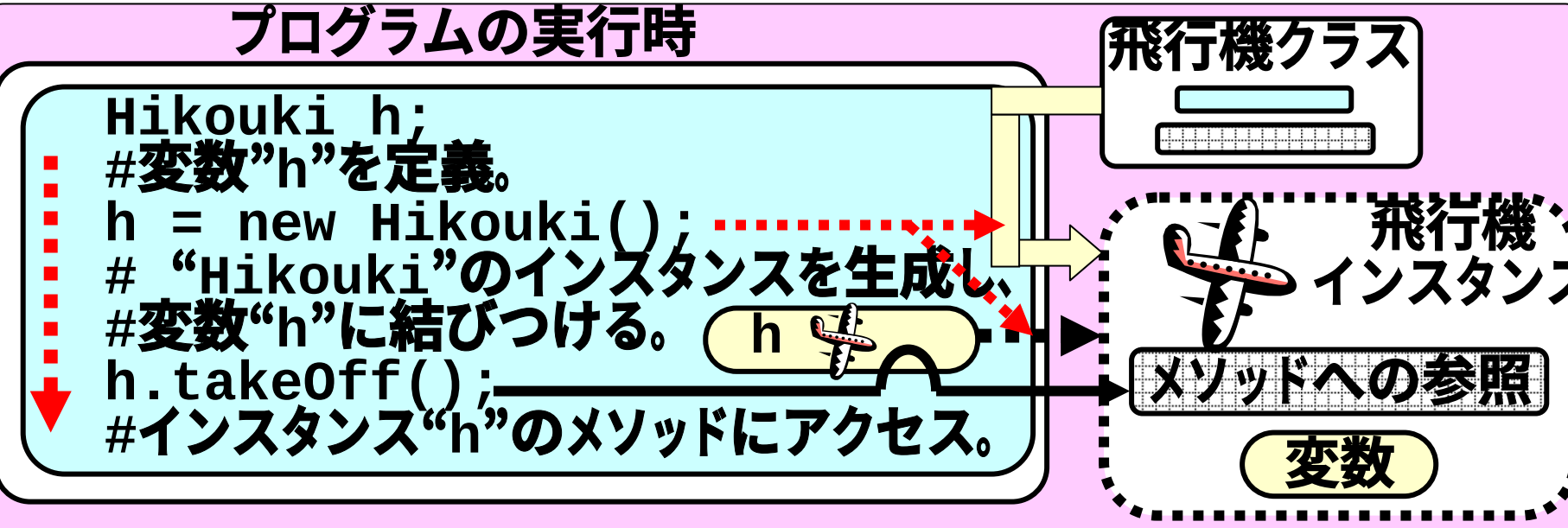


(6．2．3) Javaでのインスタンス

■Javaでは、次のようにインスタンスを生成します。

```
# 生成を行う側
        :
        :
Hikouki h;
h = new Hikouki();
h.takeOff();
        :
```

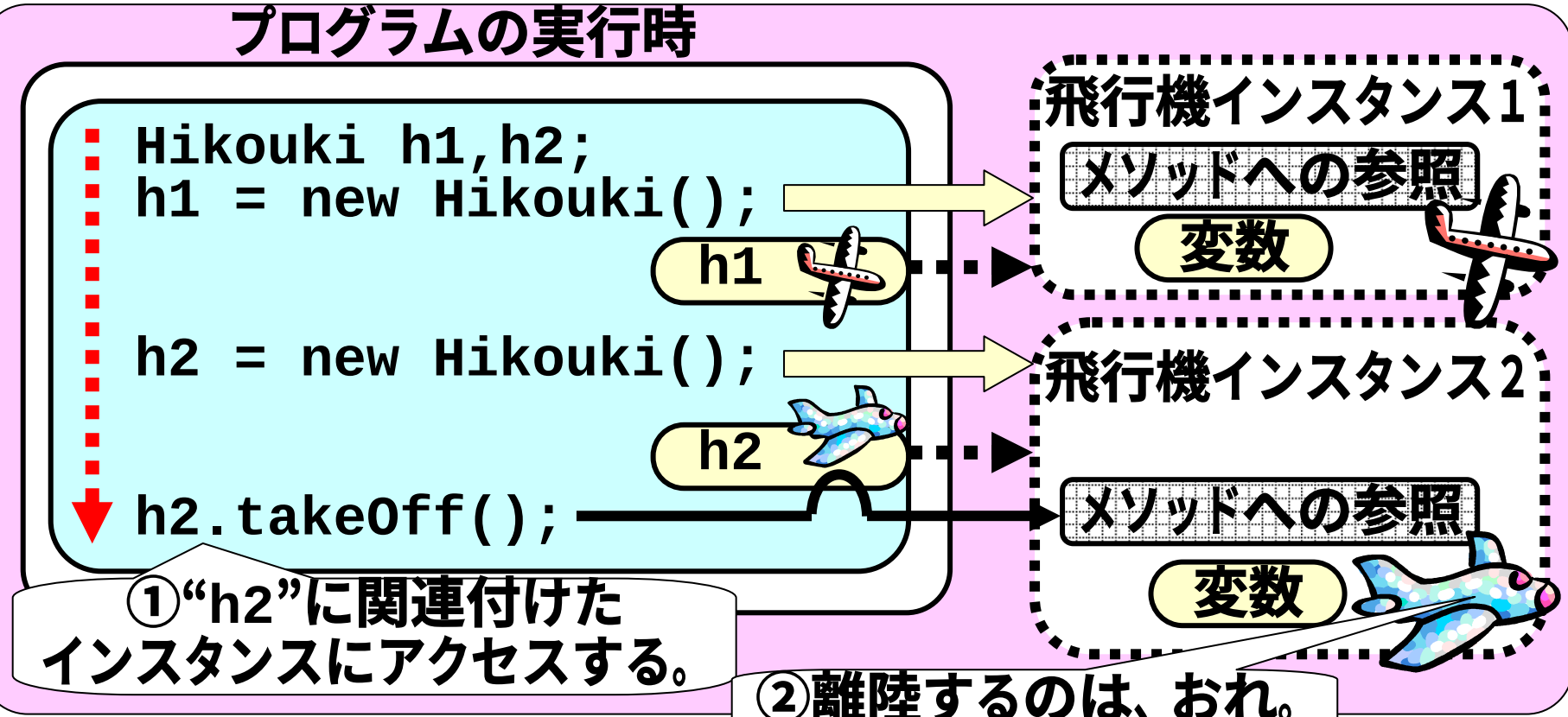
```
# 生成されるクラス
class Hikouki{
  private int sokudo;
  public void takeOff(){
        :
  }
}
```



(6．2．4) インスタンスの識別

■それぞれを区別するインスタンスを2つ生成する場合、インスタンスに結びつける変数も2つ必要になります。

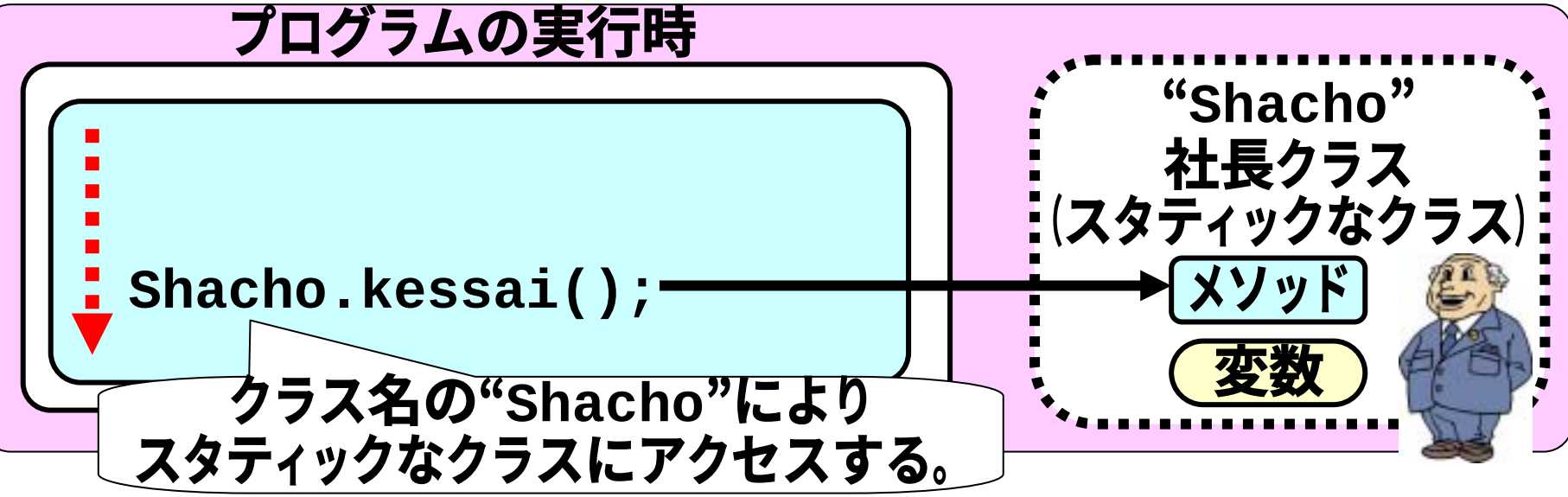
■どちらのインスタンスにアクセスするかは、どちらの変数を通じてメソッドを呼び出すかにより決まります。



(6．2．5) スタティックなクラスへのアクセス

■スタティックなクラスへのアクセスは、クラス名を使って行います。

- スライド(6.1.3)で見たとおり、スタティックなクラスは、システムで1つだけですので、他のクラスと紛れることはありません。
- スライド(6.2.1)で見たとおり、スタティックなクラスは、プログラムの実行時にはすでに存在しているので、生成する必要はありません。



(6．2．6) スタティックの指定

■Javaでは、スタティックの指定を、“static”という修飾子を用いて次のように行います。

```
# スタティックなクラス
class Shacho{
   private static int kigen;
   public static void kessai(){
            :
   }
}
```

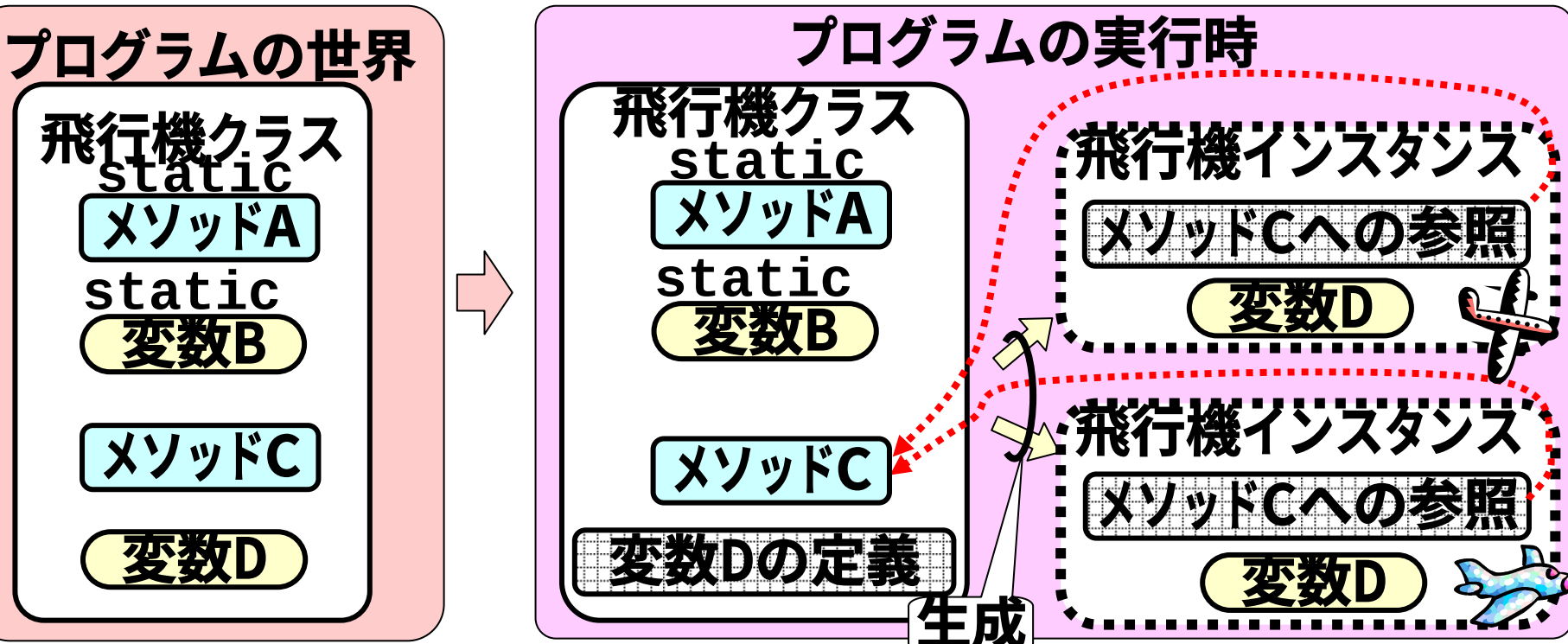
```
# インスタンスが生成されるクラス
class Roudousha{
   private int gekkyu;
   public void haitatsu(){
            :
   }
}
```

■Javaオブジェクト指向スライド(5)

(6．2．7) スタティックとインスタンス

■ここまで、「スタティックなクラス」という言い方をしてきましたが、これはあまり正確な言い方ではありません。

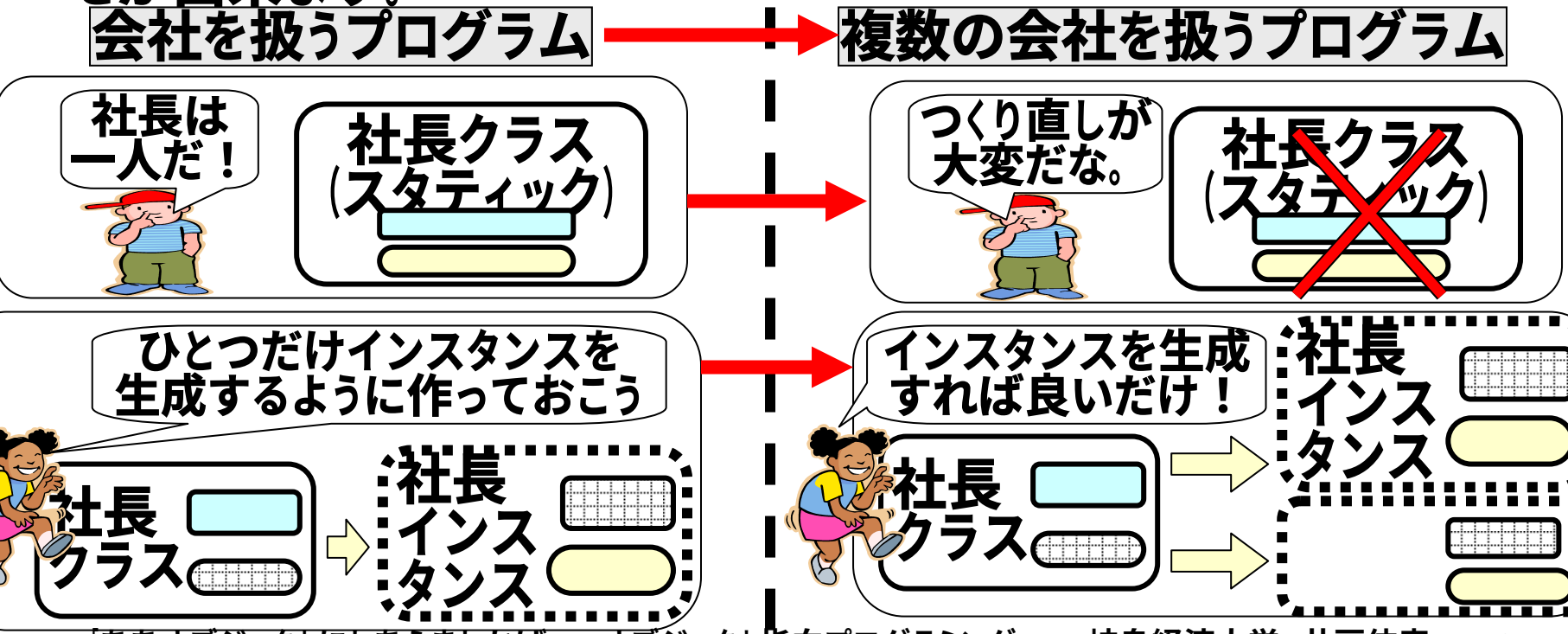
■“static”の修飾は、変数とメソッドごとにつきます。すなわち、1つのクラスで、スタティックな部分とインスタンスを発生する部分とを持つことも出来ます。



(6．2．8) なるべくインスタンスとする

■どちらかと言えば、スタティックなクラスは避ける傾向にあります。

■“私が作るシステムにこのクラスはひとつしかない”と思っていても、他のシステムにそのクラスを適用した場合には2つ以上になることもあります。“インスタンスをひとつしか生成しないクラス”として作っておけば、そのような場合にも簡単に適用することが出来ます。

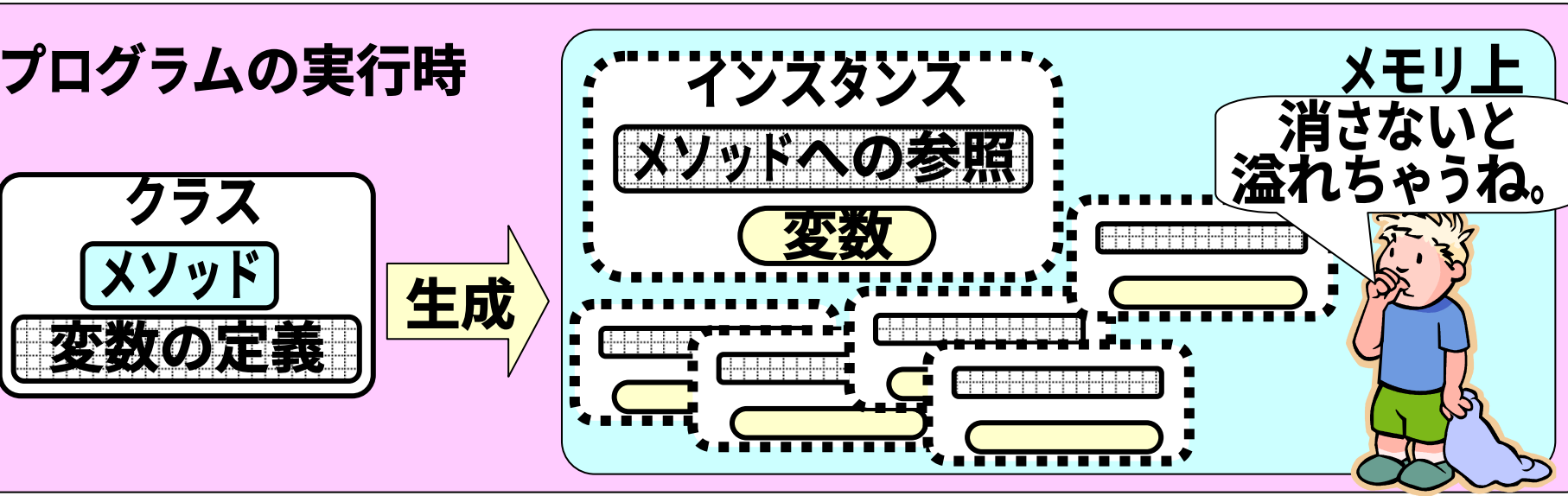


(6．2．9) インスタンスの消去

■インスタンスを生成した時には、元となったクラスの定義にあった変数が、実際にメモリ上に取られます。これが増えていけば、いずれメモリは一杯になります。

■プログラムで使わなくなったインスタンスのメモリは、これを解放してやる必要があります。

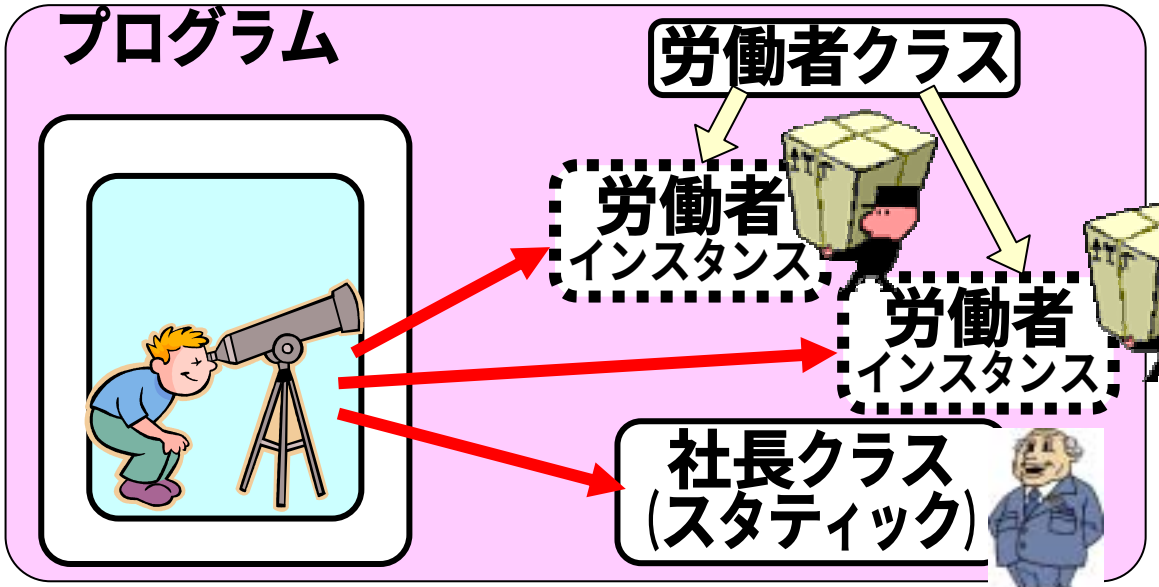
■Javaでは使わなくなったメモリの解放を自動で行っています (これをガベッジコレクションといいます)。



(6．2．10) 識別性

■スライド(2.1)に示したとおり、オブジェクトの性質として識別性があります。あるオブジェクトを別のオブジェクトと区別出来る性質のことです。

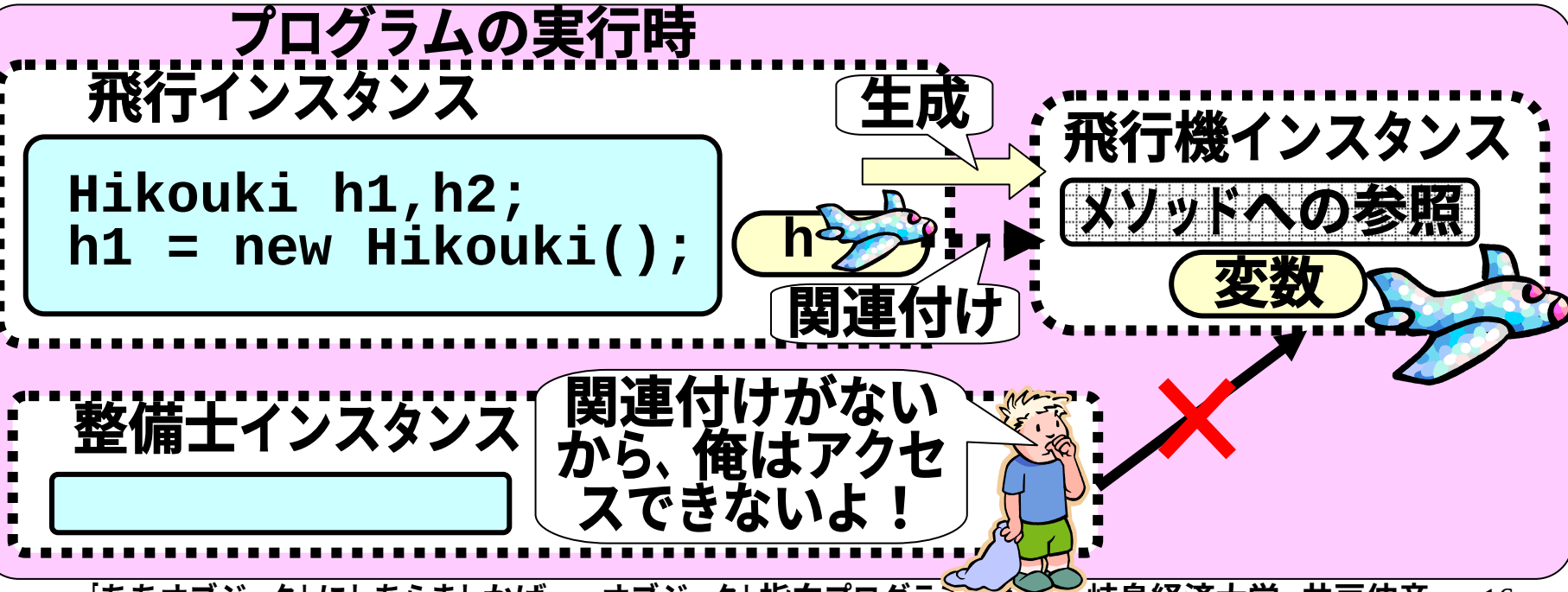
■プログラムの世界で“識別する”ことは、すなわち、他のオブジェクトと区別して、あるオブジェクトのメソッドを起動出来る、もしくは、変数にアクセス出来ることを指します。



(6．2．11)インスタンスを呼ぶ場合

■インスタンスを呼ぶ場合、インスタンスに関連付けた変数の値がわからないと、アクセスできません。

■下図の例で言うと、飛行機インスタンスに整備士インスタンスからアクセスしようとしても、整備士インスタンスには飛行機インスタンスへの関連付けが解らないのでアクセスできない訳です。

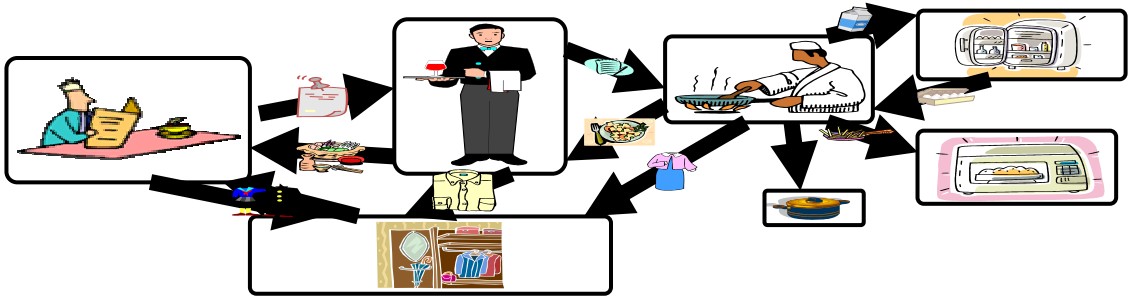


(6．2．12) どうやって識別するか？

■飛行機インスタンスと関連付けられた変数“h”を持つ飛行インスタンスが、整備士インスタンスへの関連付け (図中では変数“s”) を持っていて、“h”の値を整備士インスタンスに知らせてやれば、その値(“h”のコピー)を使って整備士インスタンスから飛行機インスタンスへアクセスすることが出来るようになります。

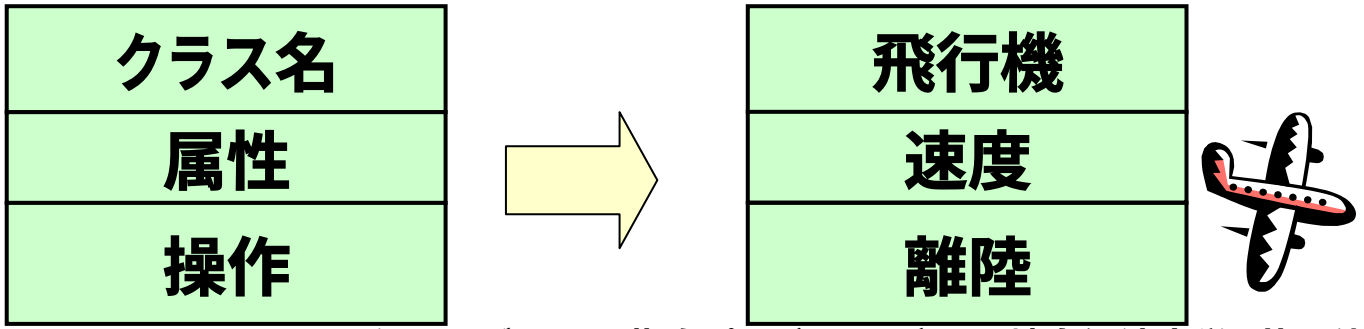
(6．3) UMLのクラス図

■UMLでは、クラス図を描いて、プログラムに登場するクラスについて、①その内容と、②互いの関係とを整理します。



■UMLクラス図におけるクラスの内容の記述は、至って素直なものです。

■ひとつのクラスは、3段に仕切られた四角で表され、上段から下段へ、①クラス名、②属性、③操作をそれぞれ記述します。Javaで言えば、①クラス名、②データ、③メソッドとなります。



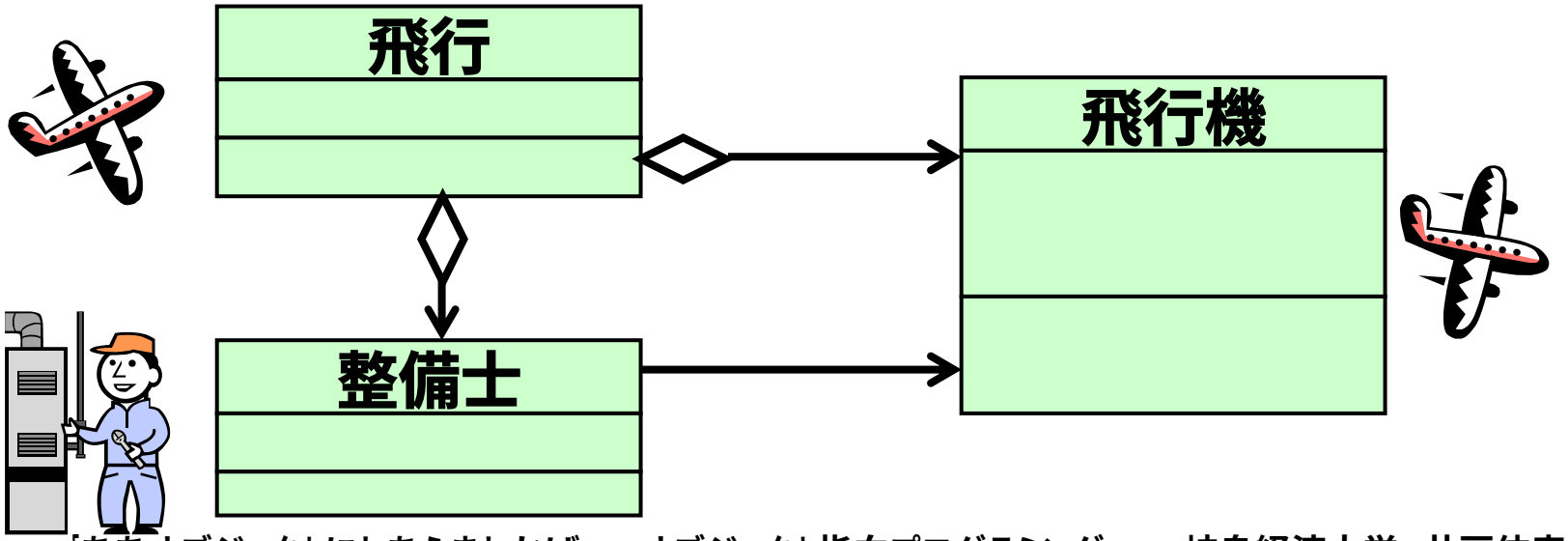
(6．3．1) クラスの関係の記述

■例えば、スライド(6.2.10)の場合を考えると、飛行／飛行機／整備士のクラスは、互いに関係を持っていることがわかります。

■クラス図では、下図のような記法で記します。

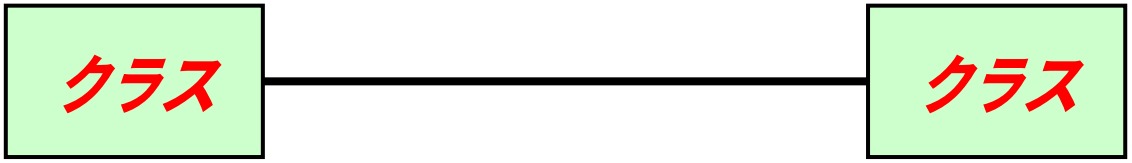
■このようなクラスの関係を、集約 (aggregation：アグレゲーション) と呼びます。

■次のスライドで、もっと一般的な記法を見てみます。

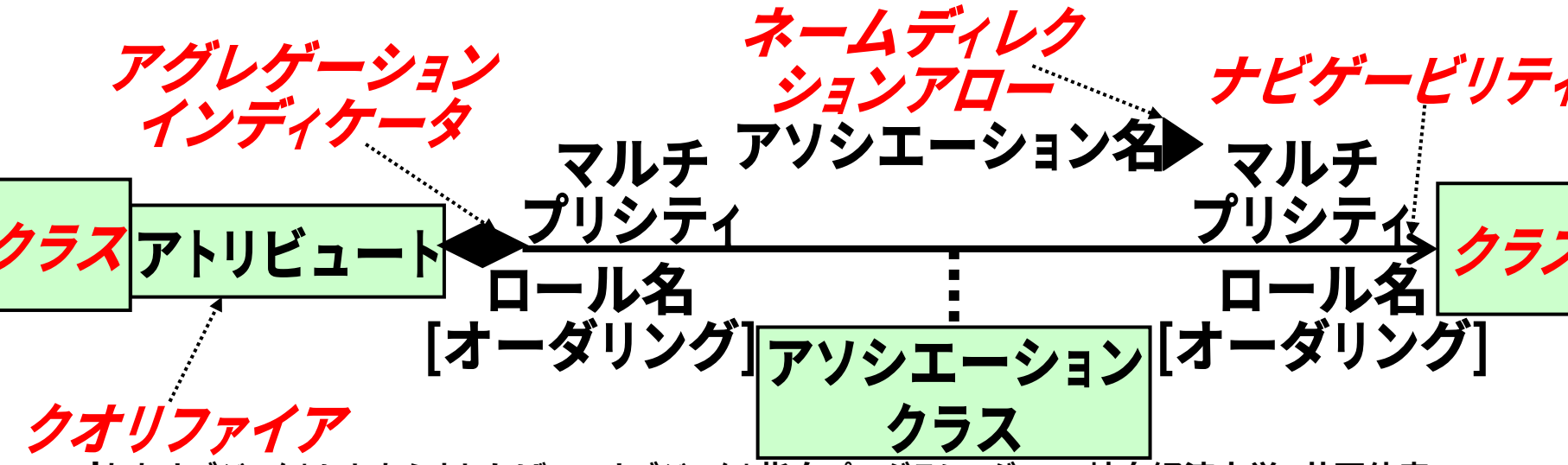


(6．3．2) アソシエーション

■関係のあるクラスを線で結ぶことは自然な記法です。これをアソシエーションと呼びます。



■UMLでは、アソシエーションの意味を表現するため、さまざまな修飾の記法があります。集約(アグレゲーション)も、そのひとつです。



(6．3．3) アソシエーション

■アソシエーションの修飾は、次の3つに分かれます。
- ・アソシエーション名、・アソシエーションロール、
- ・アソシエーションクラス

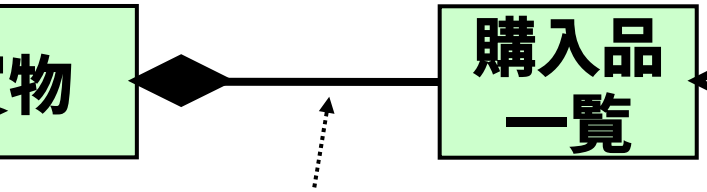
■アソシエーションロールには、次の6つがあります。
- ・アグレゲーションインジケータ、・ナビゲービリティ
- ・クオリファイア 、・マルチプリシティ、・ロール名、
- ・オーダリング

■省略可能なものもあり、参考書により省略の度合いが異なる場合など、煩雑に感じられるかもしれません。

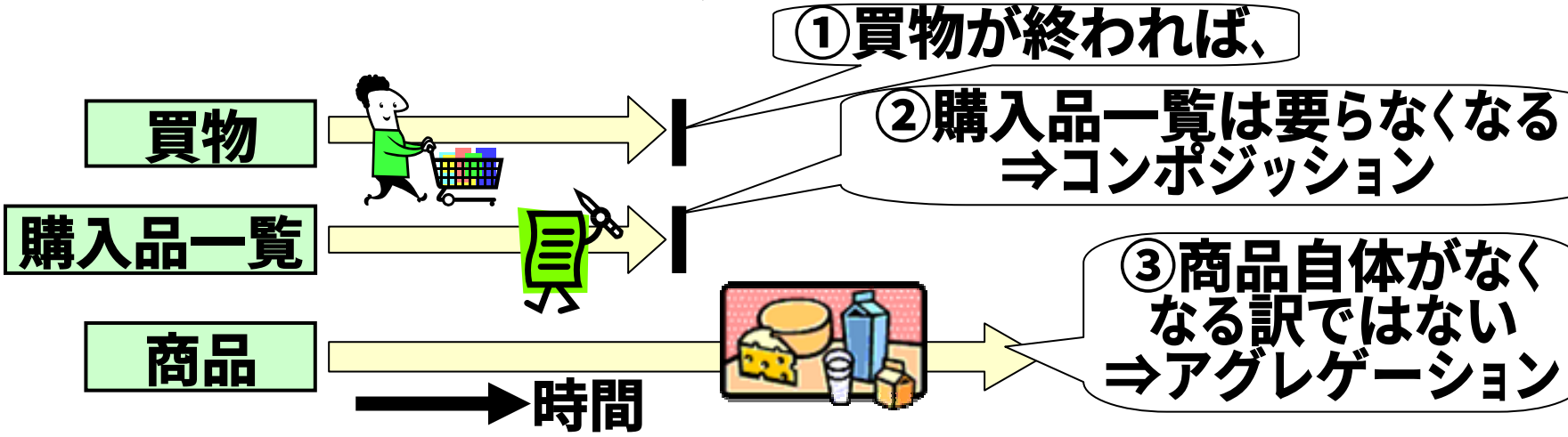
■ここでは、基本的なものについて説明していきます。

(6．3．4) アグレゲーション (集約)

■アグレゲーションには、黒と白抜きとの2種類があります

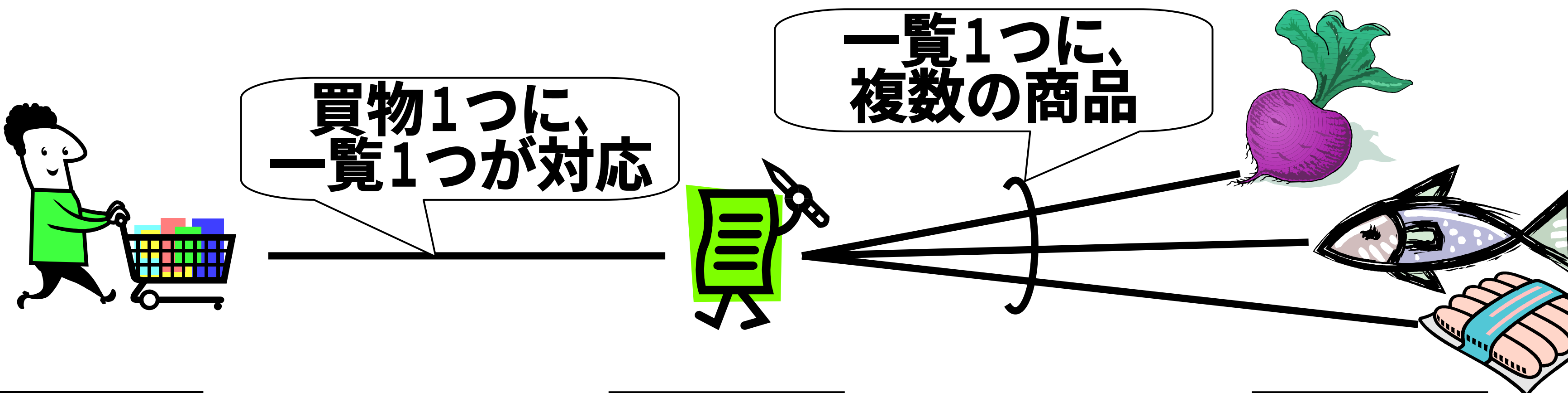


■いずれも「ひし形側のクラスに、逆のクラスを**所有している**」という意味を持ちます。コンポジッションは、さらに、「ひし形側のクラスが消滅すれば、逆のクラスのクラスも消滅する」という強い結びつきを示します。



(6．3．5) マルチプリシティ (多重度)

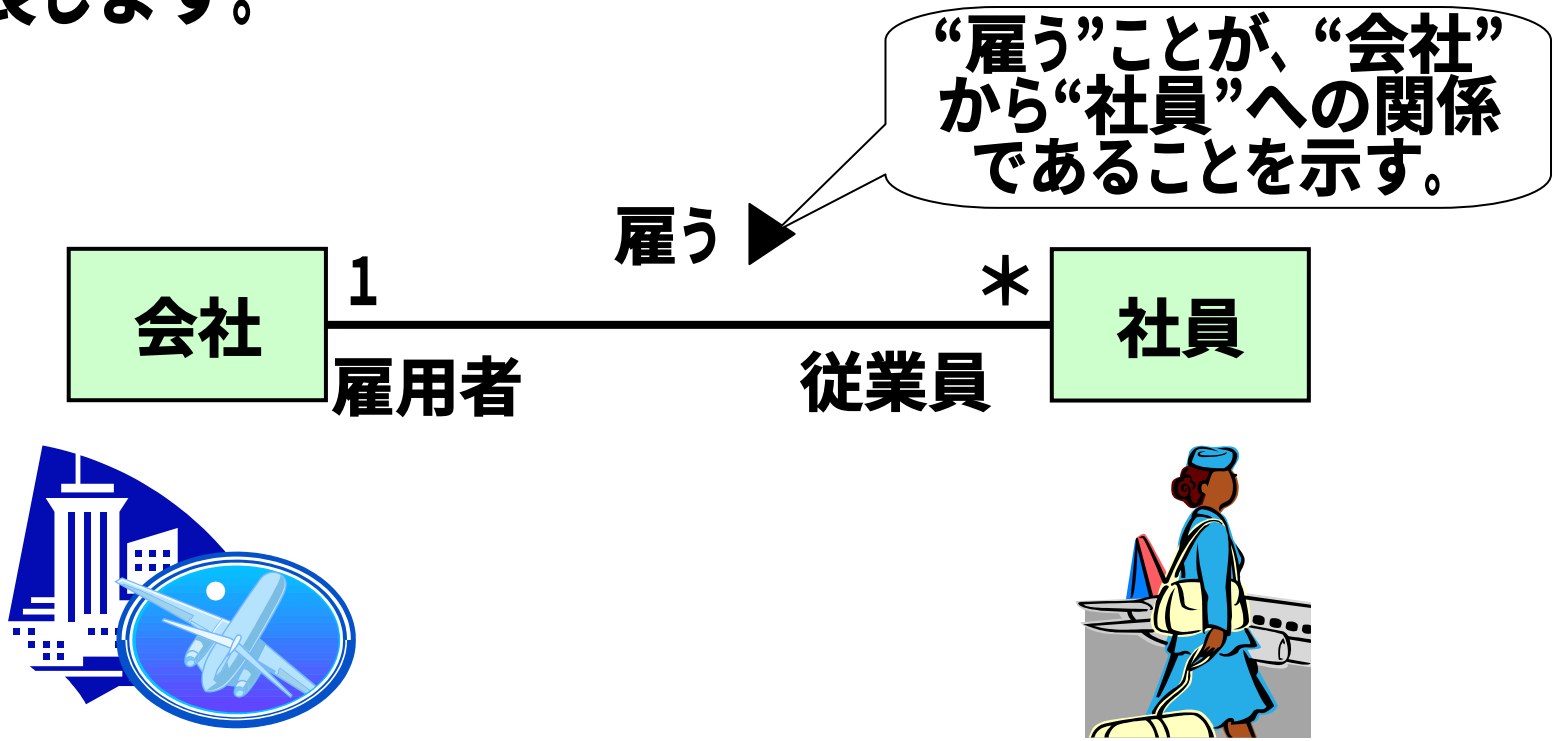
- マルチプリシティは、対応するインスタンスの数を示します。
- 買物一回につき、購入品一覧は1つです。
- 購入品一覧には、商品は0個以上対応します。
  - “＊”は、“制限なし”を表します。
  - “1．．3”は、1から3個であることを表します。



(6．3．6) アソシエーション名、ロール名

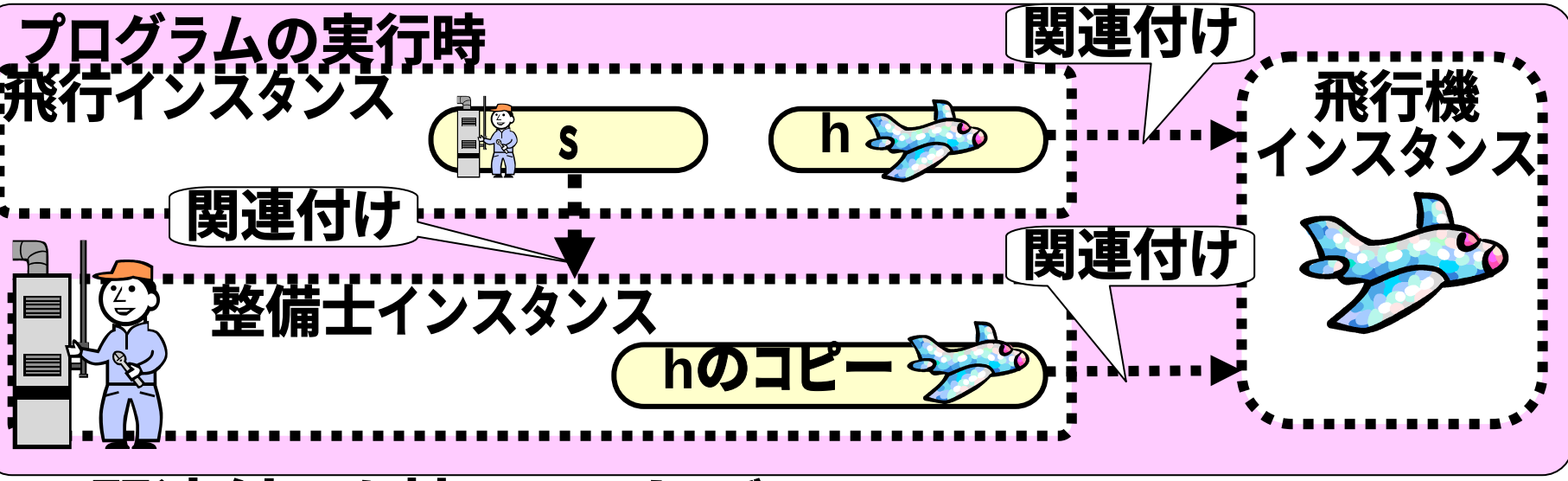
■アソシエーション名には、アソシエーションの意味づけを記します。

■ロール名は、クラスにとってのアソシエーションの役割を表します。

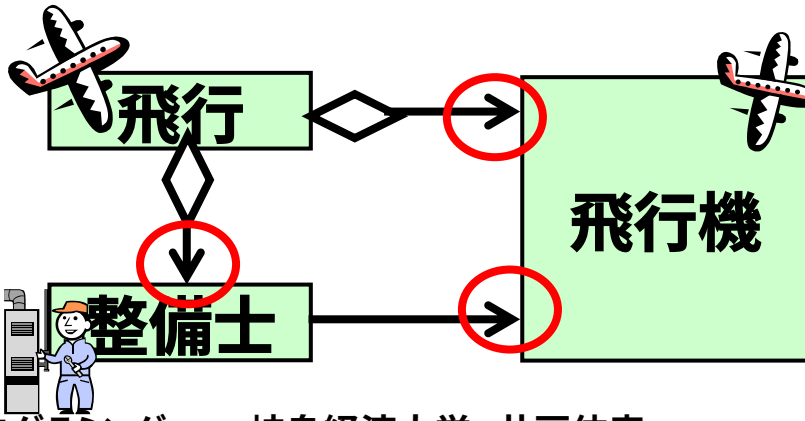


(6．3．7) ナビゲービリティ

■ナビゲービリティとは、スライド(6．2．12)で説明したような、関連付けのデータを持っていることを示します。

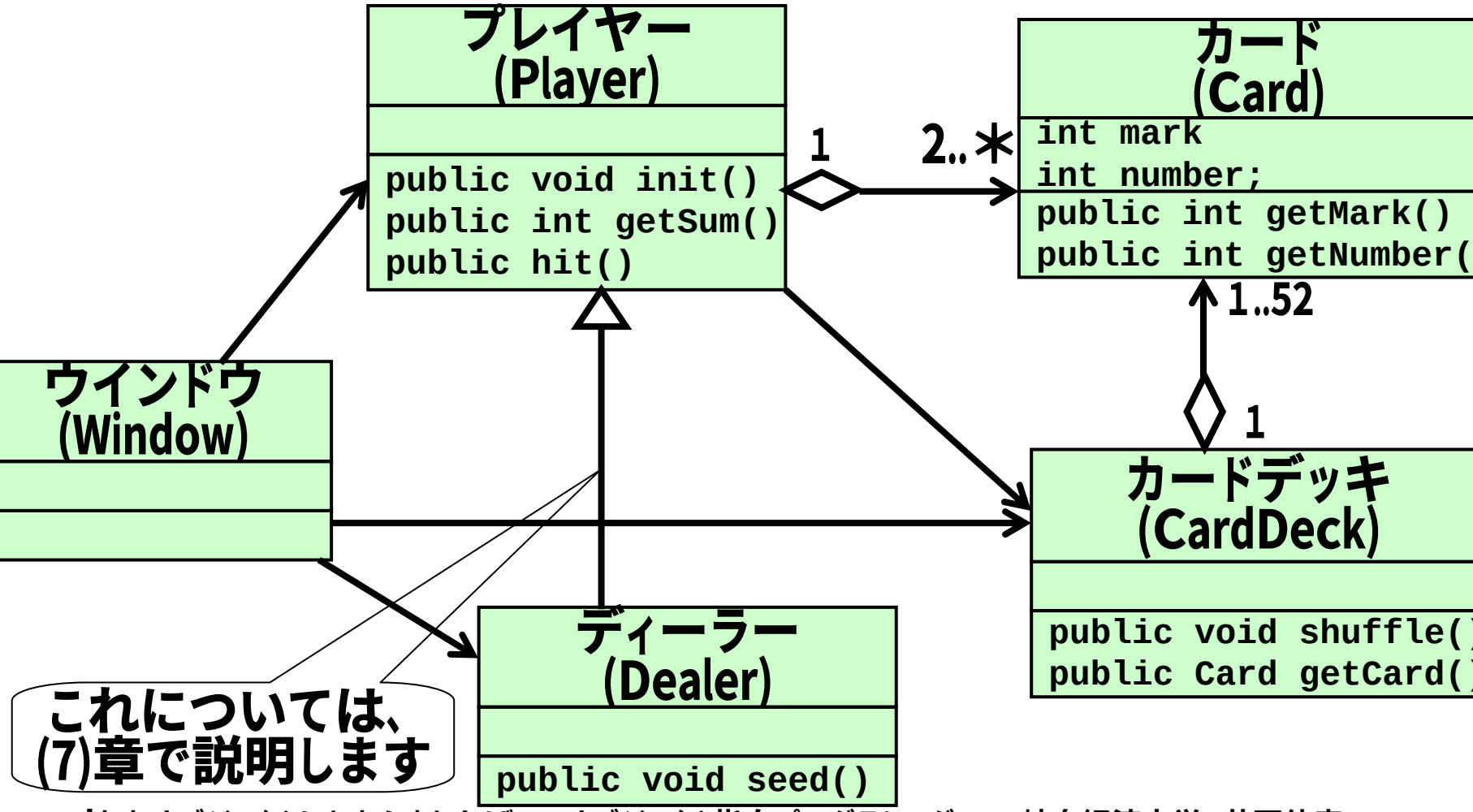


■関連付けを持っていれば、
そのクラスを参照出来る
ことになりますから、
参照可能性(ナビゲービリティ)
と呼びます。



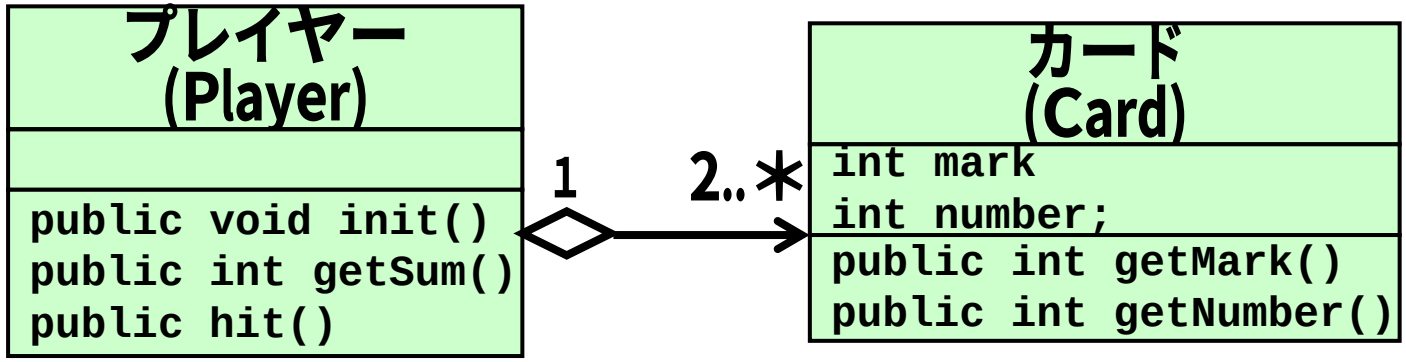
(6．4) ブラックジャックのクラス図

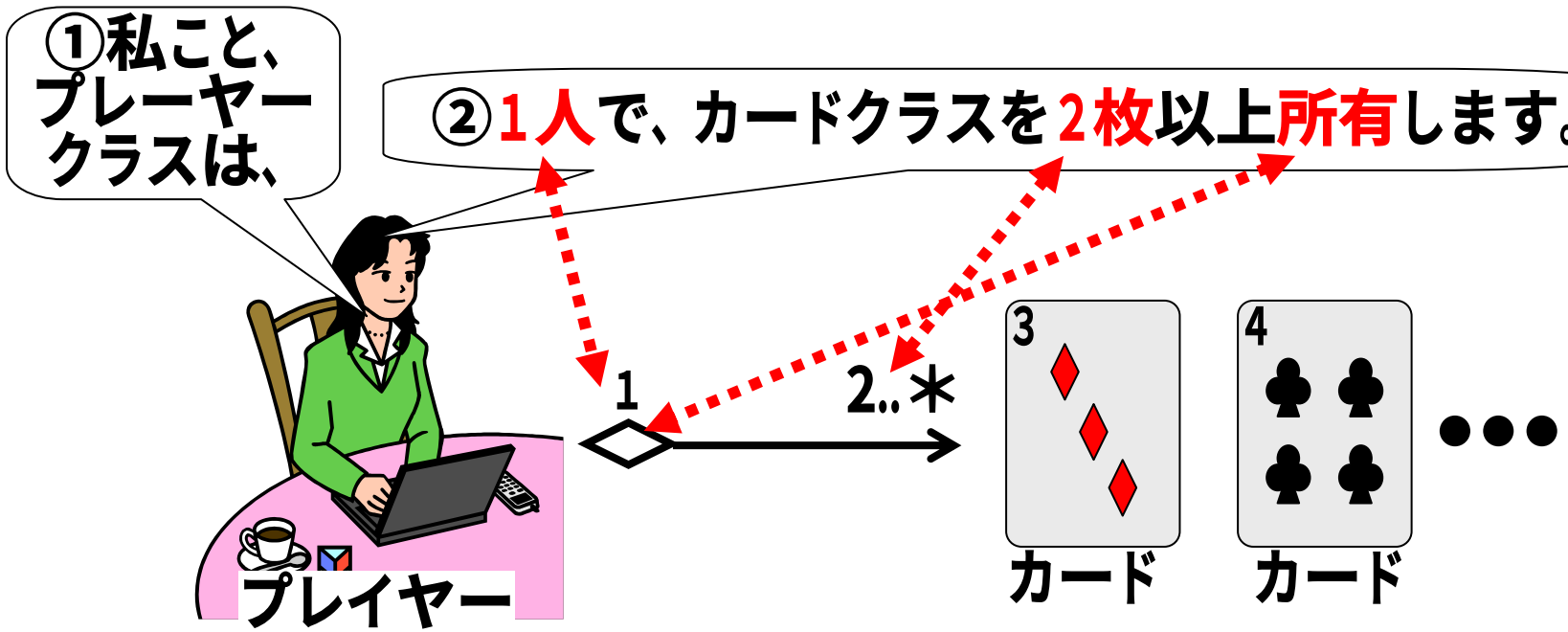
■ブラックジャックのクラス図を、次に示します（まだ未完成）。



(6．4．1) プレイヤー、カード

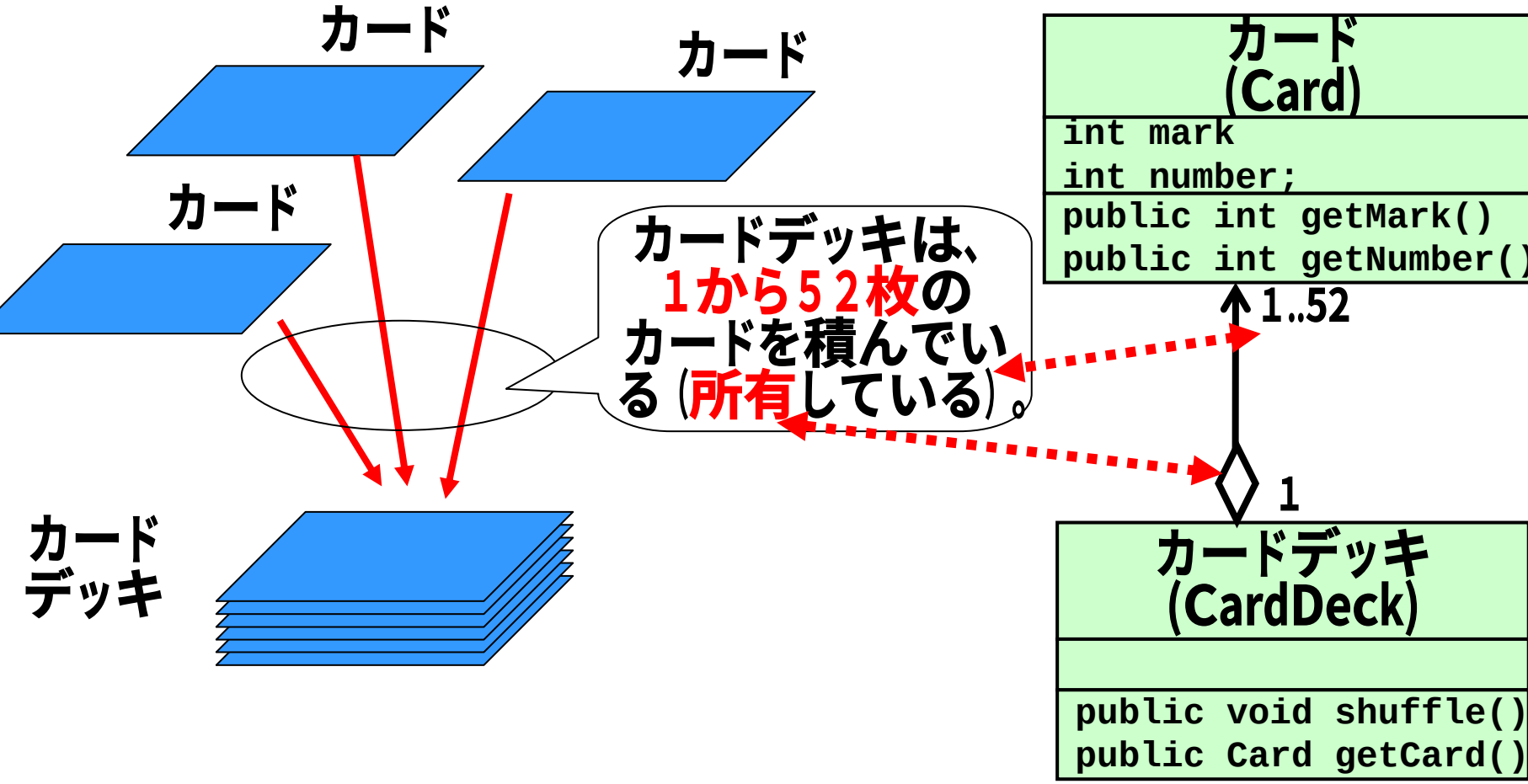
■プレーヤークラスとカードクラスとの関係を見てみましょう。





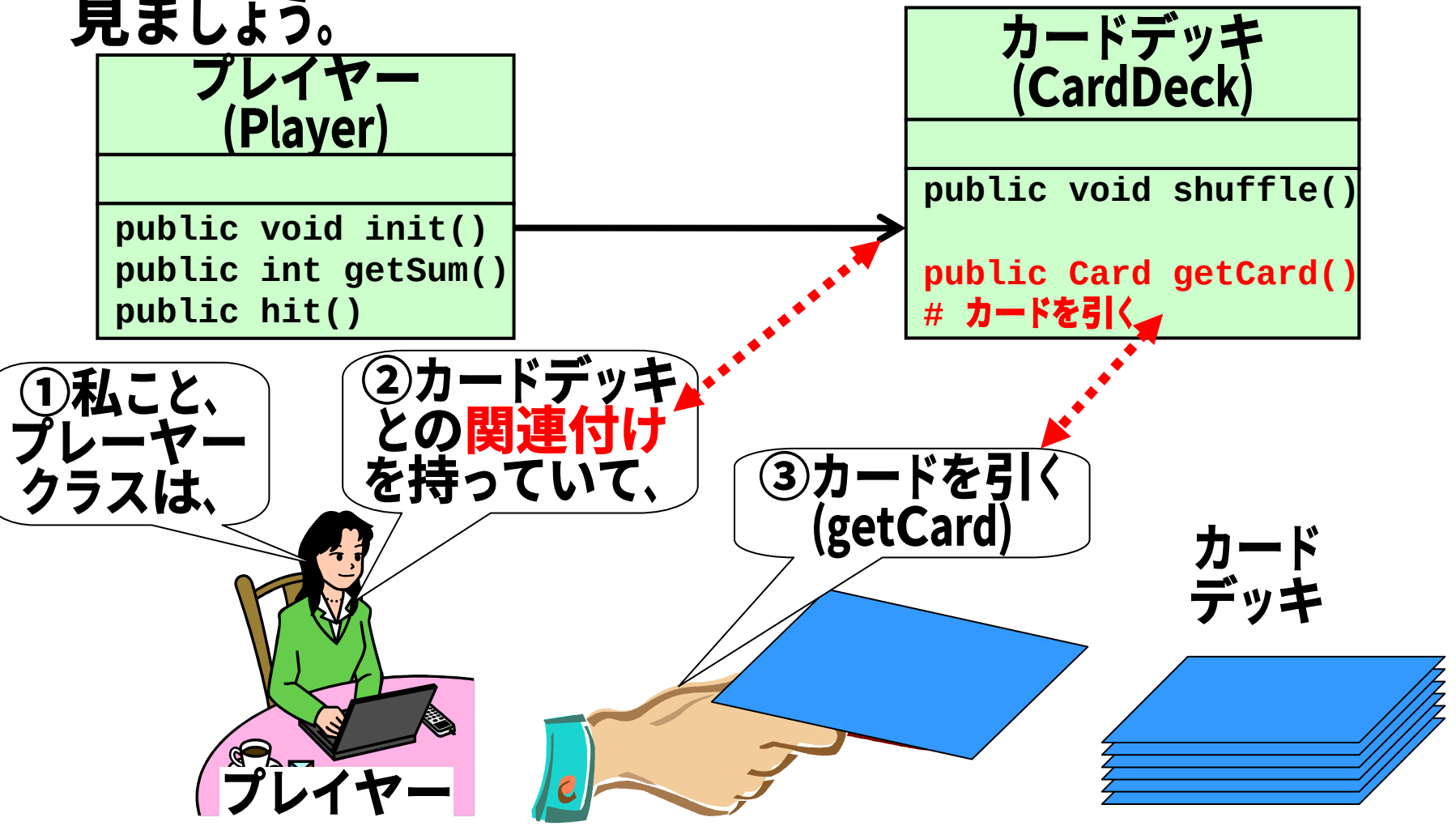
(6．4．2) カードデッキ、カード

■カードデッキクラスとカードクラスとの関係を見て見ましょう。



(6．4．3) プレイヤー、カードデッキ

■プレイヤークラスとカードデッキクラスとの関係を見て見ましょう。



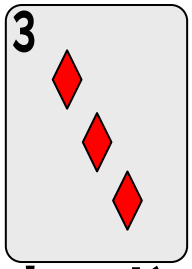
(6．4．4) プログラムとの対応　－1－

■プレイヤーには、“list”という変数があります。

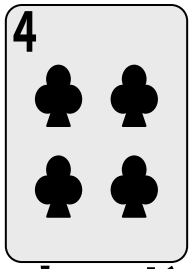
■このには、カードクラスのインスタンスが蓄えられています。

プレイヤー

1 ◇——→ 2..＊



カード



カード

●●●

①私こと、プレーヤークラスは、

```
# プレイヤークラス
public class Player {
  // Attributes
  Card[] list;
```

②1人で、カードクラスを2枚以上所有してます。

list

カードクラスのインスタンス ♦ 3

カードクラスのインスタンス ♣ 4

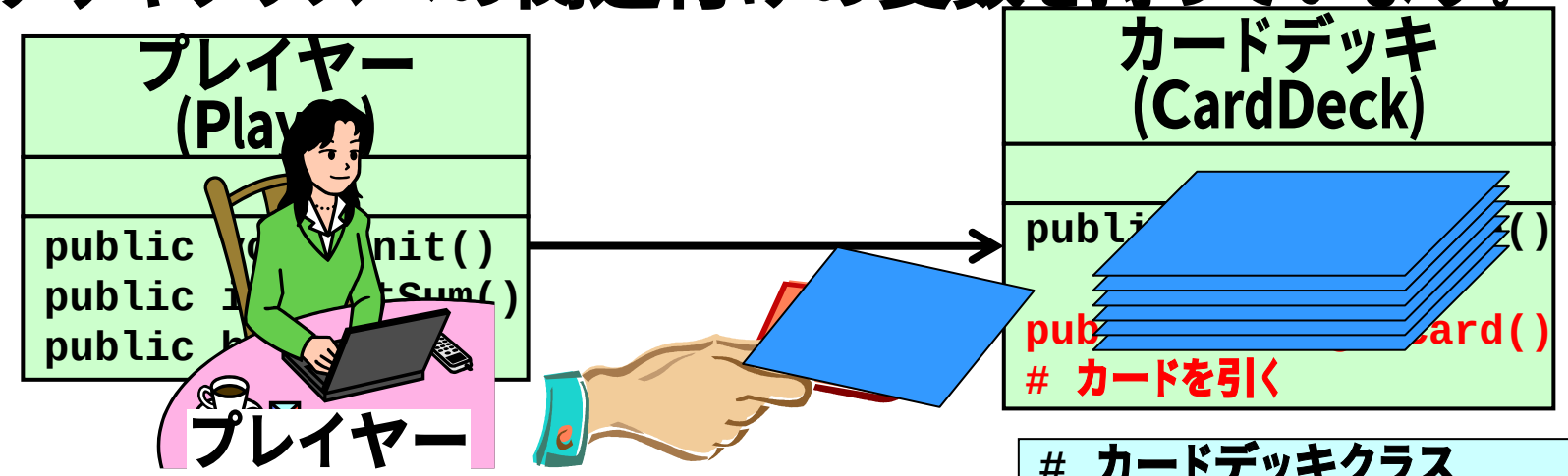
```
# カードクラス
public class Card {
  private int mark;
  private int number;

      :
```

■クラス型変数⇒Javaオブジェクト指向スライド(6)

(6．4．5) プログラムとの対応　－２－

■プレイヤークラスには、“myCardDeck”という、カードデッキクラスへの関連付けの変数を持っています。




```
# プレイヤークラス
public class Player {

  // Attributes
  private CardDeck myCardDeck;
      :
protected void addOne(){
      .....myCardDeck.getCard().....
  }
```

①私こと、プレーヤークラスは、

②カードデッキとの関連付けを持っていて、

③カードを引く(getCard)

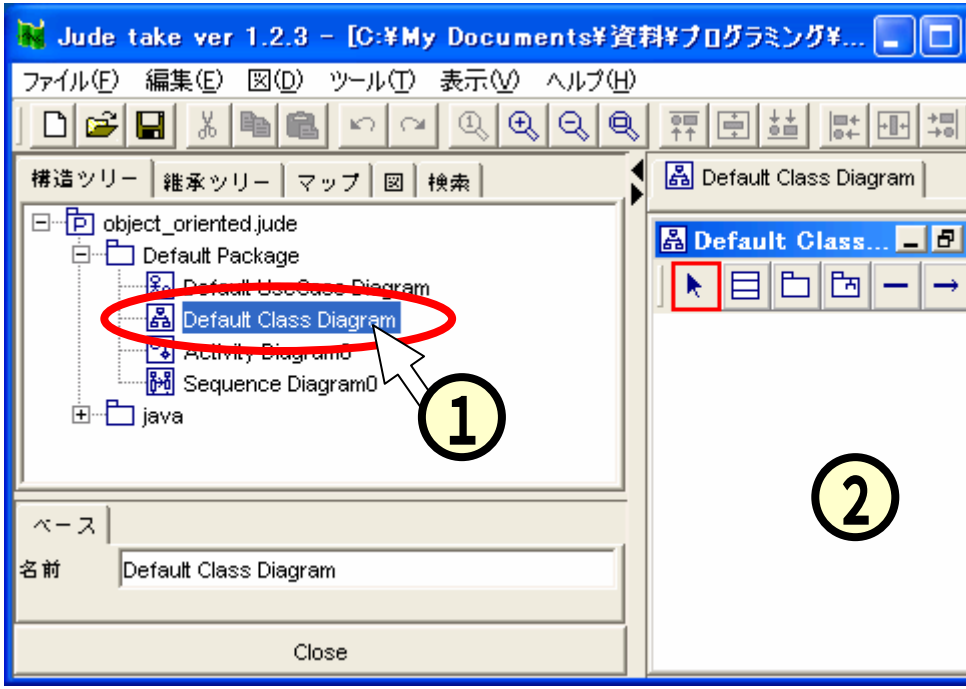
```
# カードデッキクラス
public class CardDeck {
      :
ic Card getCard(){
      :
```

(6．4．6) 演習：課題6－1、6－2

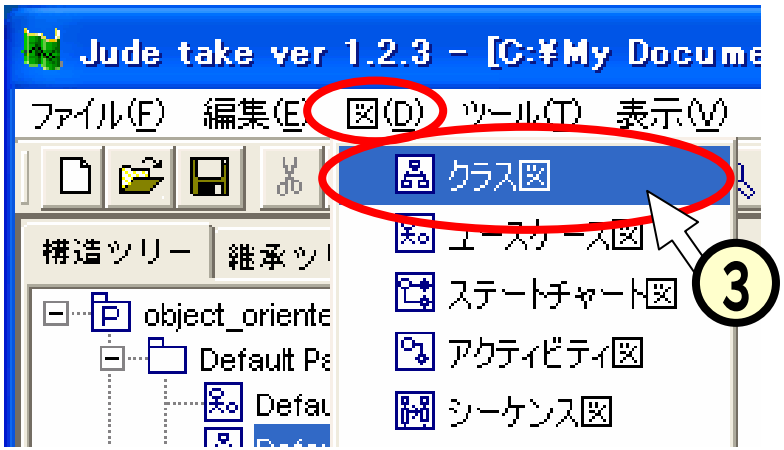
- 演習1と同様に、“ブラックジャック”のプログラムを、井戸のネットワークドライブからダウンロードしてください。
- (課題6－1) ダウンロードしたプログラムを見て、スライド(6.4.4),(6.4.5)の内容を確認してください。
  - プログラム中にコメントで位置が示されています。
- (課題6－2) ダウンロードしたプログラムを見て、スライド(6.4.2)の内容にあたる部分を探してください。
  - そこを抜き出して、井戸までメールしてください。

(6．5．1) 新規クラス図を開く

■プロジェクトビュー中の”Default Class Diagram” をクリック (①) してフォーカスする (薄紫色になる) と、編集画面 (②) にて図の作成が可能になります。

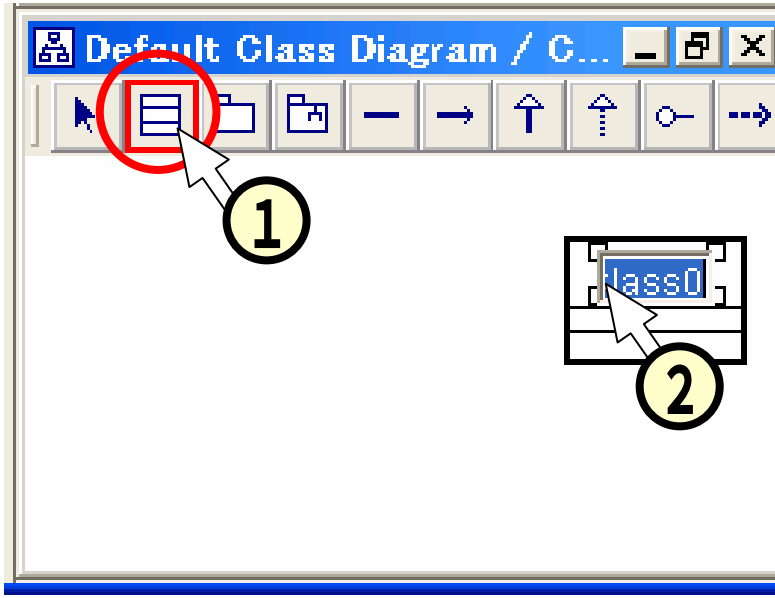


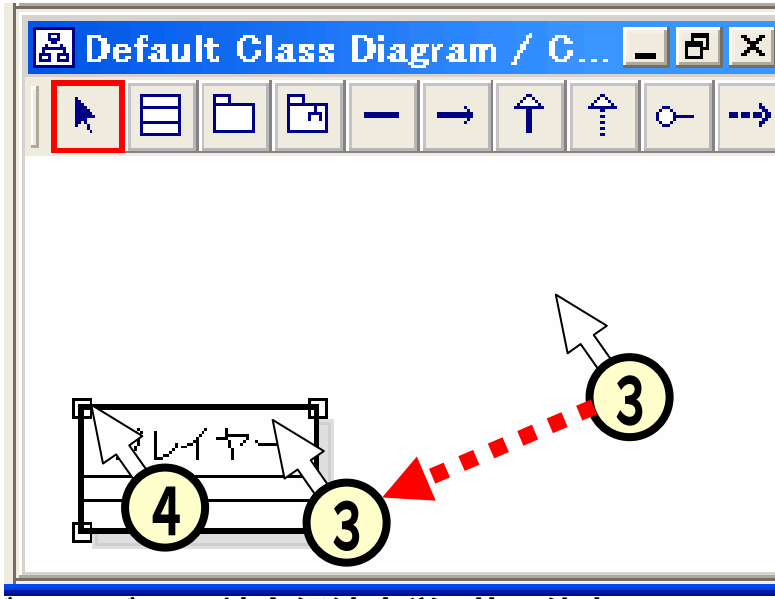
■新たなクラス図を作成する場合には、[図]-[クラス図]をクリック (③) します。



(6.5.2) クラス要素の配置

- ツールバーの[クラス]をクリック(①)して、ダイアグラム･エディタ上をクリック(②)します。
- クラス要素が、クラス名の入力待ちの形で配置されます。ここで名前の入力が可能ですが、後から名前の欄をダブルクリックして入力することも出来ます。
- ドラッグ(③)することで、移動させます。また、要素の四隅のつまみをドラッグ(④)して、大きさを調整します。
- フォーカスされた状態(＝四隅につまみが表示)で[Delete]キー押下すると、要素は削除されます。





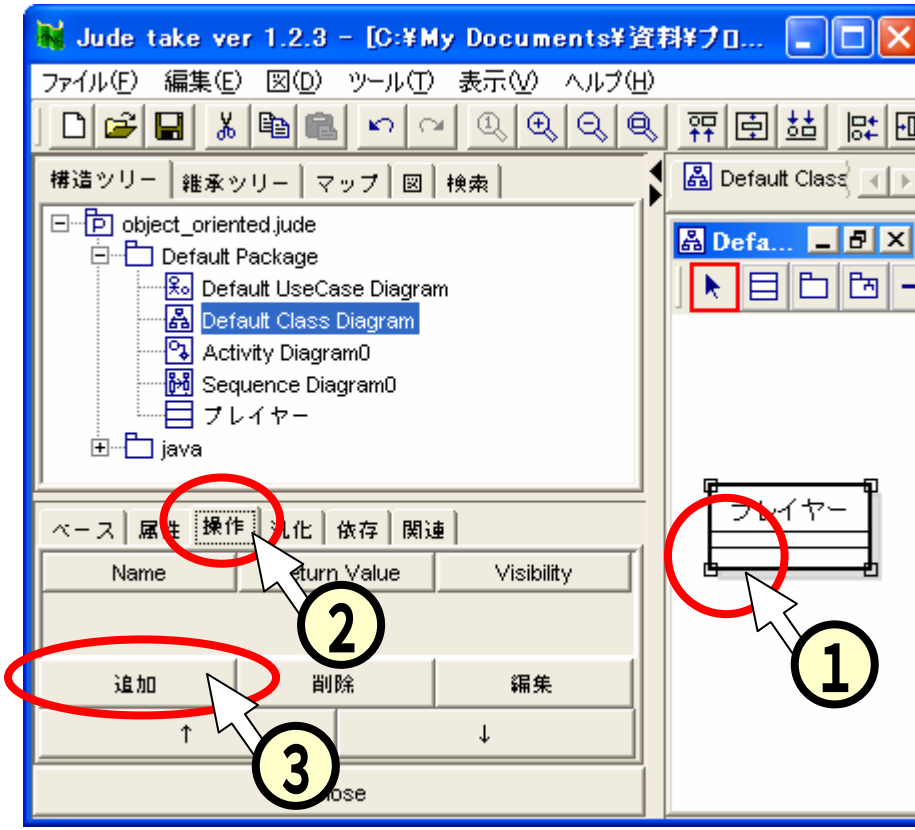
(6．5．3．1) 操作(属性)の編集　－1－

■クラス要素をクリック(①)して、フォーカスします。

■プロパティ･ビュー中の[操作]タグをクリック(②)します。

■[追加]ボタンをクリック(③)します。

■新たな操作の行が追加され、この行をクリックして選択(④)した状態で、編集ボタンをクリック(⑤)します。



(6．5．3．2) 操作 (属性) の編集　－2－

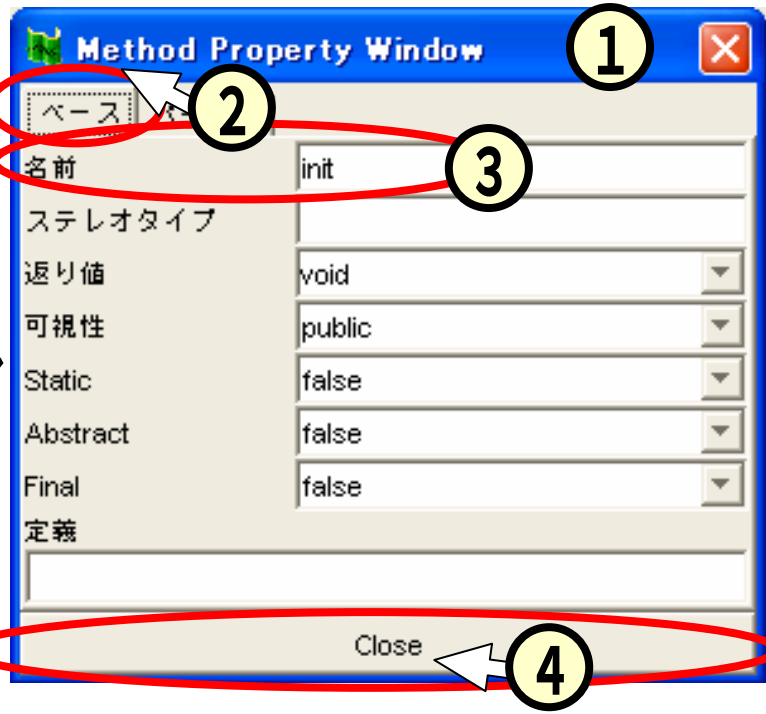
(前スライドから続く)

■「Method Property Window」(①) が開きます。

■ベースタグをクリック (②) して、名前 (③) などを入力します

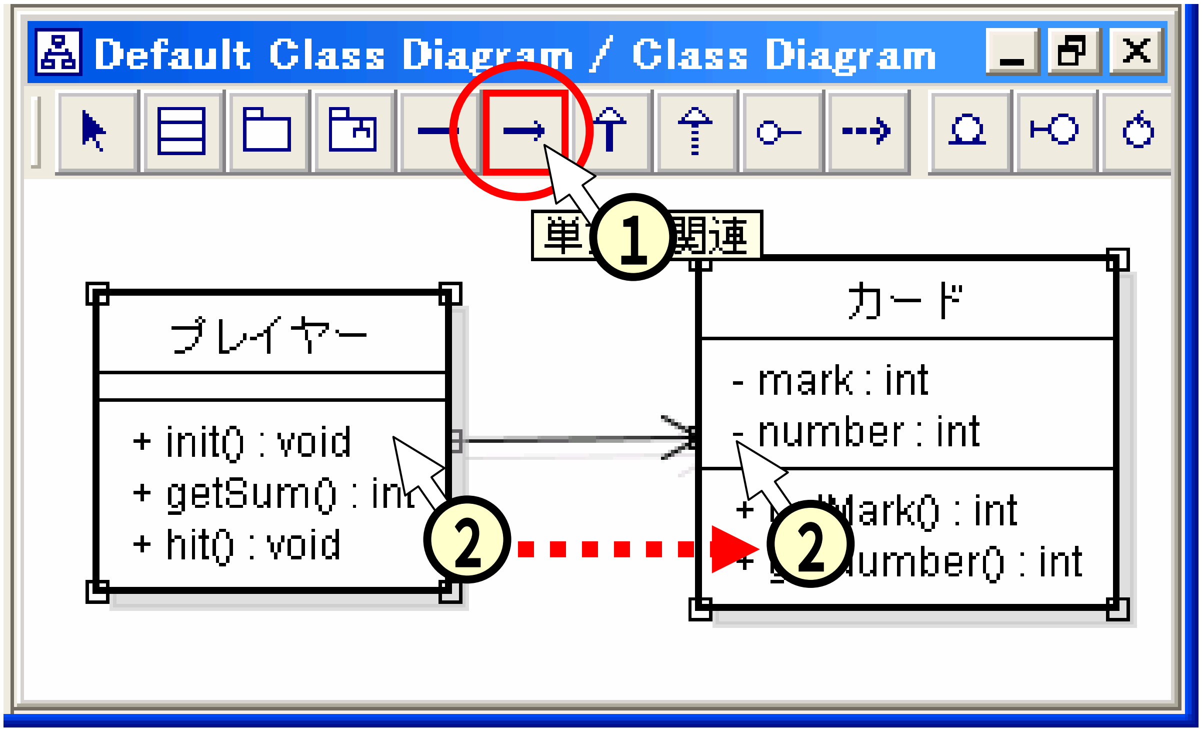
■[Close]ボタンをクリック (④) すると、属性が入力されています (⑤、⑥)。

- 今回は、他の項目の設定は行わなくても結構です。
- また、パラメタの設定についても、同様です。
- 属性の編集についても、操作の編集とほぼ同様に行います。



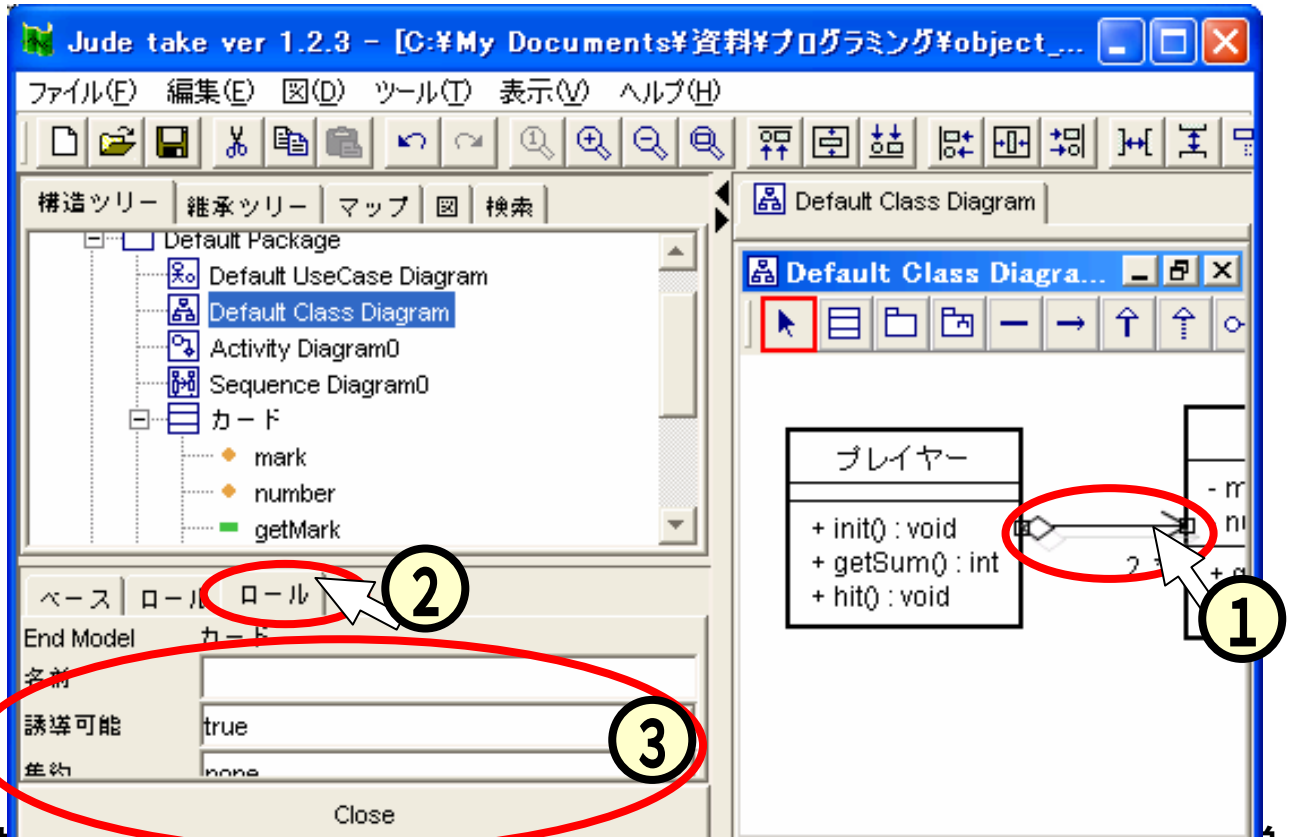
(6．5．4) アソシエーション要素の配置

■ツールバーの[単方向関連]をクリックし（①）、アソシエーションの元側（プレイヤークラス）からアソシエーションの先側（カードクラス）へと、ドラッグします（②）。



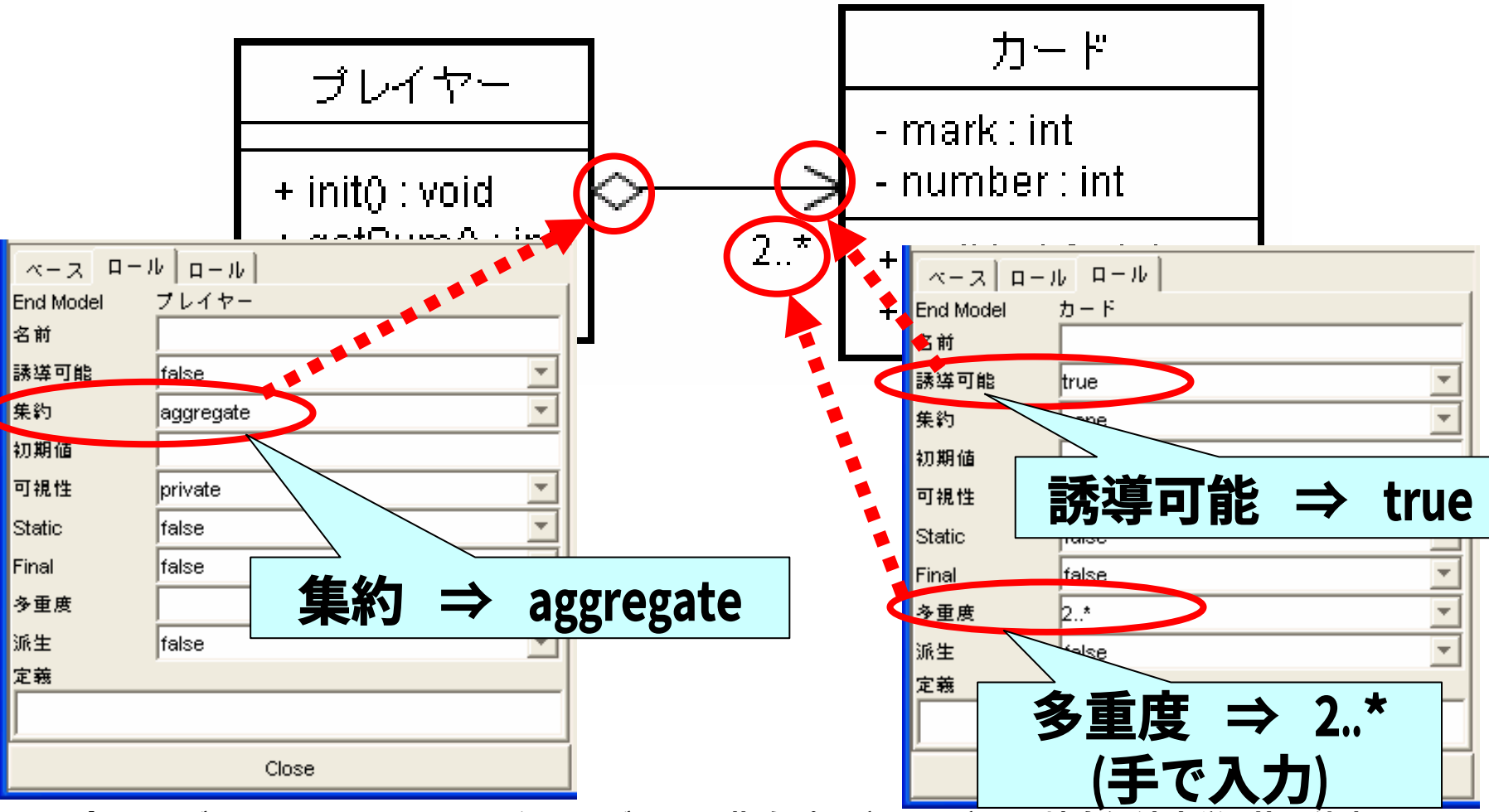
(6．5．5．1) アソシエーションの修飾ー1ー

■アソシエーションを選択した状態 (①) で、プロパティ・ビュー中の[ロール]タグをクリック (②) して、アソシエーションの修飾を行います (③) (以下はプレイヤーとカードの例です)。



(6．5．5．2) アソシエーションの修飾ー2ー

■プロパティ･ビューでの設定と、アソシエーションの修飾とは、次のように対応します。



(6．5．6) 演習：課題6－3、6－4

■課題6－3：スライド(6.4)のクラス図を描いてください。

- 属性、操作については、省略してもOKです。

■課題6－4：スライド(5.7.2)～(5.7.5)演習5で考えた、さいころゲームのクラス図を描いてください。