オブジェクト指向技術 第3～5回
― モデリング 静的モデル―

3回目keyword:クラス図，属性、操作、関連、継承
4回目Keyword:抽象クラス，多態性、動的束縛
5回目Keyword: 集約と複合、総称

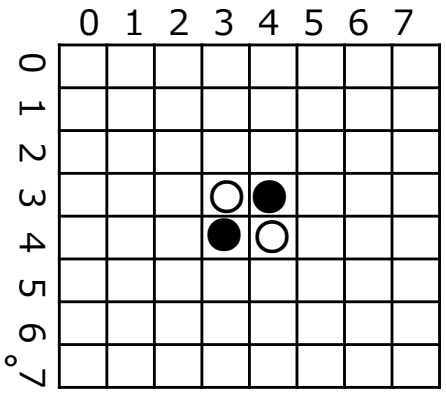
立命館大学　情報理工学部
槇原　絵里奈
Mail: makihara@fc.ritsumei.ac.jp

講義内容

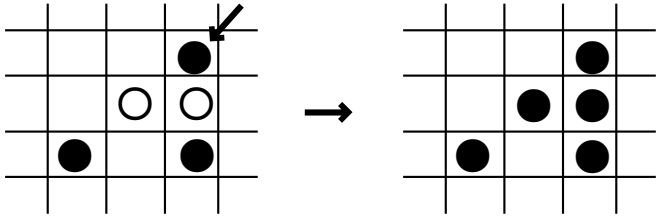
- 要求分析
  - リバーシの開発を例に
  - ユースケース図
- 静的モデリング
  - 名詞抽出法
  - オブジェクト図・クラス図
  - 属性と操作
  - 継承・汎化
  - 委譲

黒白棋

例題：リバーシゲーム(othello)の開発



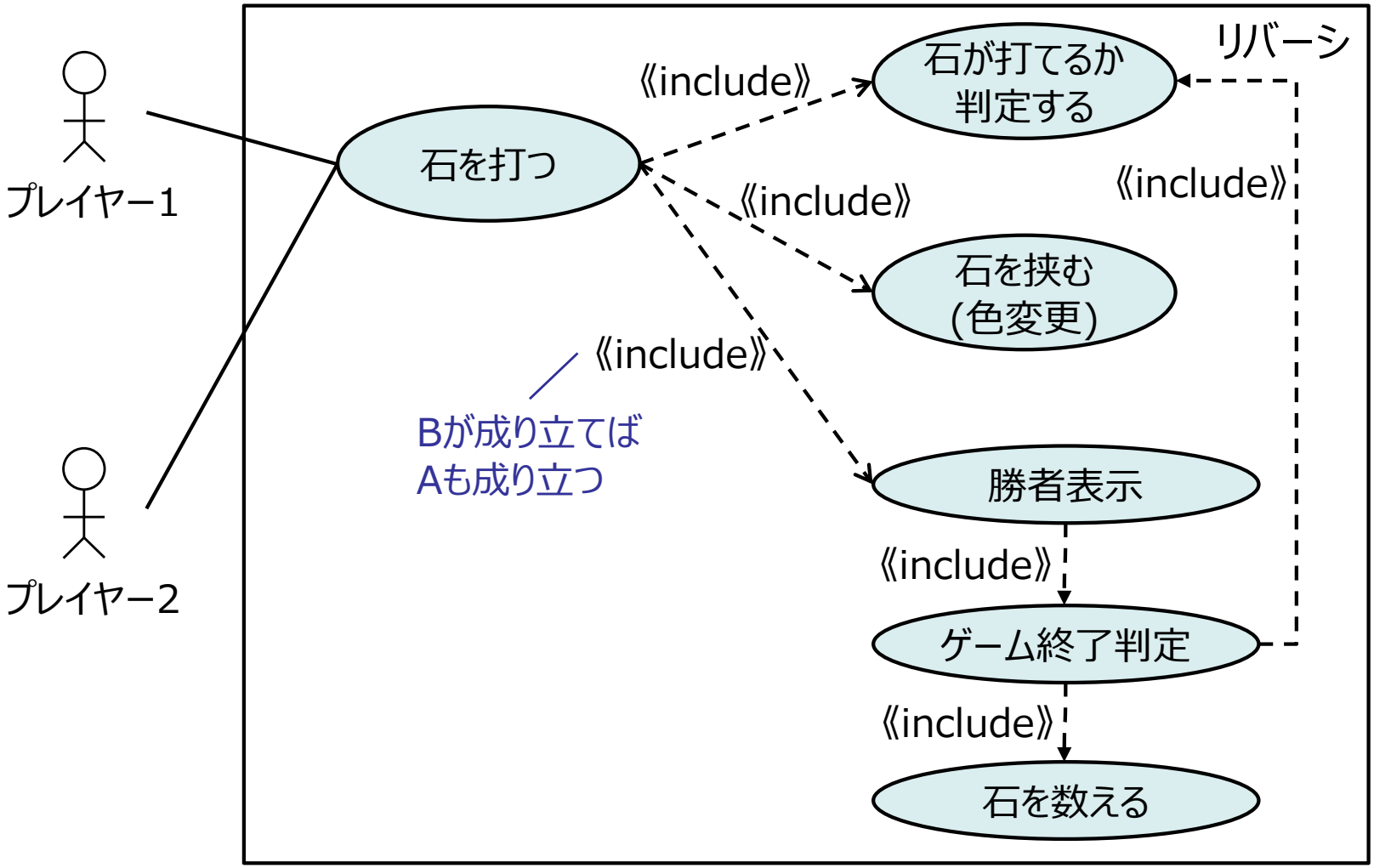
- リバーシ(Reversi)
  - 盤面は縦８マス×横８マスで構成される。
  - プレイヤーは先手、後手の２人である。
  - 先手は黒石を、後手は白石を使用する。
  - 初期の盤面は、中央に黒石と白石を２つずつ並べた状態とする。(座標(3,3)(4,4)に白石、(3,4)(4,3)に黒石を設置。)
  - プレイヤーの手番は先手から開始する。
  - プレイヤーは自分の手番に、盤面の１つの空きマスに自色の石を１つ設置する。ただし、自色の他の石で縦・横・斜め方向に敵色の石を挟めなければならない。挟むとは、設置した石自身と他の同色の石の間に隙間なく他色の石が並ぶことを意味する。
  - 石の設置後、その設置によって挟まれた敵石すべてを自色に変える。
  - 手番のプレイヤーが設置できるマスがない場合、相手プレイヤーの手番に移る。
  - ２人とも設置できるマスがない場合、ゲーム終了とする。
  - ゲーム終了時、自色の石が多いプレイヤーの勝利とする。黒と白の石が同数であれば引き分けとする。



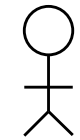
要求分析とユースケース図

システム全体のイメージの共有
ユーザ視点でこのシステムは何ができるのかを明示

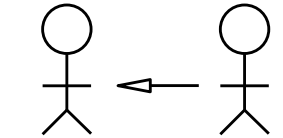
- システムの機能ごとに作成
- システムの利用者(actor)から見た使われ方を示す



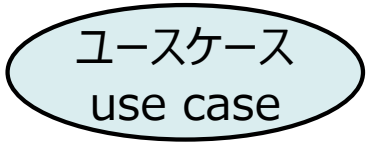
ユースケース図(use case diagram)



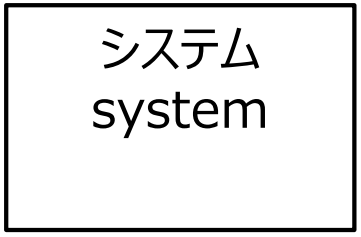
- **アクタ(actor)：システムを操作するユーザや外部のハードウェア･システム等**
  - システムに対して**利用者が果たす役割(role)**を示す
  - 役割ごとに異なるアクタが存在(同一人物であっても良い)
  - システムの場合はコンピュータ等のアイコンを使うことも可能
  - 《actor》と付記することでアクタであることを明記することも可能
    (《》で追加される記述(UML図の拡張)をステレオタイプと呼ぶ)



- **汎化：あるアクタPが他のアクタCの汎化であることを示す**
  - (例)「プレイヤー」は「人間プレイヤー」や「AIプレイヤー」の汎化
    (汎化 generalization ⇔ 特化 specialization)

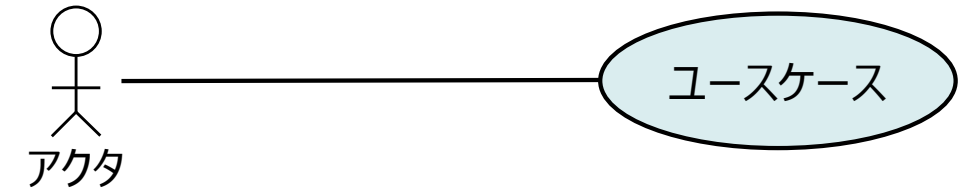


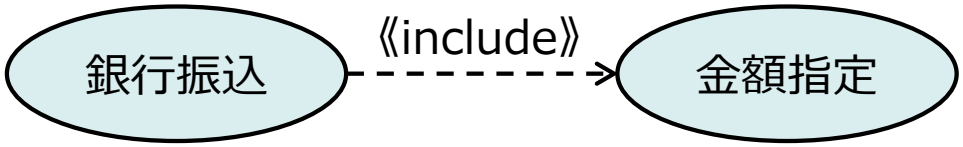
- **ユースケース(use case)**：アクタから見た**システムの機能**

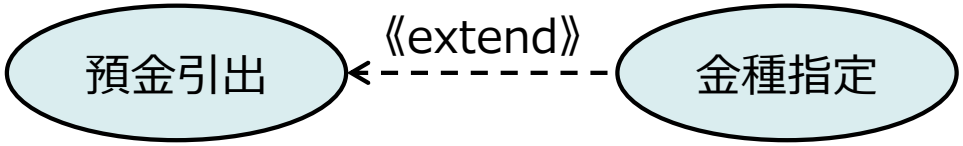


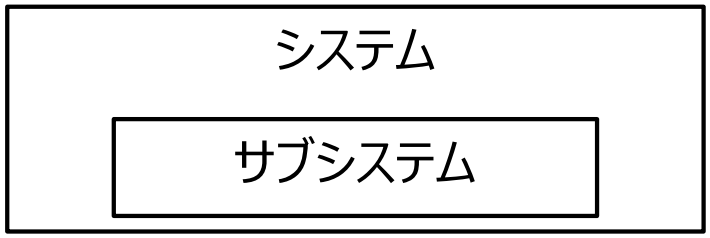
- **システム境界**：システムの内部と外部を区別
  内側にシステム名を記述
  - 内部にユースケースを記述。
  - 外部にアクタを記述。

ユースケース図(use case diagram)









- アクタとユースケースの関連
  - アクタがユースケースを実行
  - アクタがユースケースからの反応を得る
- 包含(include)
  - ユースケースが他のユースケースを内部に含む
- 拡張(extend)
  - ユースケースを別のユースケースによって拡張
- サブシステム(sub-system)
  - システムが他のサブシステムを内部に含む

イベントフロー(event flow)の作成

- ユースケースの内容を**アクタとシステムの対話形式で通常の文章で表現**
- 顧客にも理解できるように**専門用語は使わない**
- イベントフローの種類
  - **基本フロー**：アクタとシステムがユースケースを実行していくステップ
  - **代替フロー**：基本フローの中で起こりうる、通常以外のステップ
  - **例外フロー**：想定される例外とその対応

ユースケース記述

■ユースケース記述

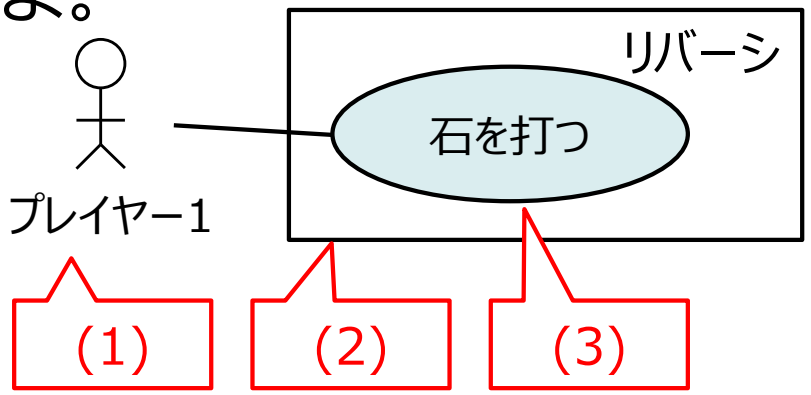
- ●イベントフローを具体的に記述したもの
- ●一般的に、**シナリオ(scenario)**により記述
- ●利用者とシステム間の対話を表す一連の手順
- ●正常系(基本フロー)だけでなく
  異常系(代替・例外フロー)も記述
- ●前提条件や事後条件も記述できる

(例) ユースケース記述

- 名称：石を打つ
- 開始アクタ：プレイヤー1、プレイヤー2
- 目的：盤面を更新し、挟まれた敵石の色変更や勝者表示を行う
- 事前条件：手番プレイヤーにより空きマスが指定済である
- 事後条件：置かれた石により挟まれた石は自色になっている
  両者打てるマスがなくなった場合は勝者が表示され、ゲームが終了している
- 正常処理シナリオ
  1. 指定されたマスに石を設置できるか判定
  2. 挟まれた敵石を自色に変更する
  3. 各プレイヤーに打てるマスがあるか判定
  3-a. (3で相手が打てる場合) 手番を相手にする
  3-b. (3で自分が打てる場合) 手番を自分にする
  3-c. (3で両者打てない場合) 各色の石を数え、勝者を表示してゲーム終了
- 例外処理
  - マスを指示した後にネットワークエラーが生じたら
    「ネットワークに異常が発生しました」と表示してゲーム終了

確認問題

- 以下の各文は正しいか。○か×で答えよ。
  - ユースケース図はシステムの処理手順を表現する。
  - ユースケース図において、システムはサブシステムを包含できる。
- ユースケースが他のユースケースを内部に含むことを示すステレオタイプは何か。
- ユースケースが他のユースケースを拡張することを示すステレオタイプは何か。
- ユースケースの詳細を明確にするため、シナリオ等により記述される説明を何と呼ぶか。
- ユースケース各部の名称を答えよ。



講義内容

■要求分析
- ●リバーシの開発を例に
- ●ユースケース図

⇨静的モデリング
- ●名詞抽出法
- ●オブジェクト図・クラス図
- ●属性と操作
- ●継承・汎化
- ●集約・複合
- ●委譲

静的モデリングの手順

- オブジェクト候補抽出
  - 要求仕様やユースケース記述からオブジェクト候補を明らかにする
- クラス抽出
  - オブジェクト候補を整理してクラスを抽出
- 属性や操作の抽出
  - クラスが持つ属性、操作を明らかにする
- クラス間関係の抽出
  - クラス間の利用関係等を明らかにする

※必ずしもこの順番でなくても良い

静的モデリングの注意点

- モデルの視点を明らかにする
  - 複数の視点を紛れ込ませない
  - 例：ユーザ視点、管理者視点

成績管理システムでは管理者
出勤システムではユーザ

- 骨格を大事にする
  - 些細な構造に囚われず、**基本構造**を明確にする
  - 細かな仕様(制御構造などコーディング)より
    大まかな仕様(誰が何をどのように使うか)
- 最小かつ完備
  - システムを実現するために必要かつ十分なモデル

オブジェクトの抽出

■ システム内のオブジェクト(もの)を明確にする

■ 名詞抽出法
- 要求仕様やユースケース記述の名詞に着目
- 名詞：物体・物質・人物・場所など具体的な対象

名詞抽出法

- リバーシ(Reversi)
  - 盤面は縦８マス×横８マスで構成される。
  - プレイヤーは先手、後手の２人である。
  - 先手は黒石を、後手は白石を使用する。
  - 初期の盤面は、中央に黒石と白石を２つずつ並べた状態とする。
    (座標(3,3)(4,4)に白石、(3,4)(4,3)に黒石を設置。)
  - プレイヤーの手番は先手から開始する。
  - プレイヤーは自分の手番に、盤面の１つの空きマスに自色の石を１つ設置する。
    ただし、自色の他の石で縦・横・斜め方向に敵色の石を挟めなければならない。
    挟むとは、設置した石自身と他の同色の石の間に隙間なく他色の石が並ぶことを意味する。
  - 石の設置後、その設置によって挟まれた敵石すべてを自色に変える。
  - 手番のプレイヤーが設置できるマスがない場合、相手プレイヤーの手番に移る。
  - ２人とも設置できるマスがない場合、ゲーム終了とする。
  - ゲーム終了時、自色の石が多いプレイヤーの勝利とする。黒と白の石が同数であれば引き分けとする。

名詞抽出法(クラスの選別)

- 盤面 (board)
- マス (cell)
- プレイヤー(player)
  → 先手を人間(human)、後手をAIとする（開発対象）
- × 手番 → 先手・後手のどちらかを指す
- × 先手 → プレイヤー
- × 後手 → プレイヤー
- × 相手 → プレイヤー
- × 石 → マスの状態と考えられる
- ゲーム → ゲーム全体（処理開始点となるクラス）

×：クラスではない(と考えられる)もの

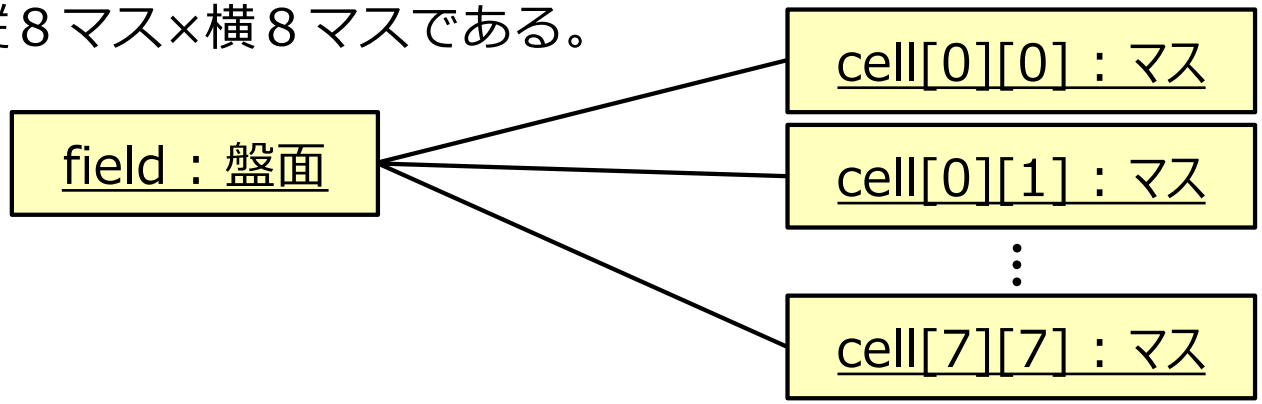
開発対象外のものはクラスではない
他のクラスと同一のもの等を除去

オブジェクト図(object diagram)

■ オブジェクトの属性と操作、オブジェクト間の関係を表現

クラスの具体的なイメージ

盤面は縦８マス×横８マスである。

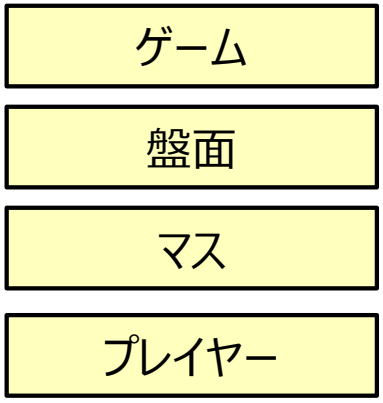


・オブジェクト名：クラス名　という書式
・オブジェクト名、クラス名のどちらかを省略してもOK
　(例　　:マス　　field:　)

クラス図(class diagram)

- クラスの**内部構造(属性・操作)**と**クラス間の静的な関係**を表現
  - オブジェクト指向分析・設計・実装をつなぐ中心的な図
- 概念的観点から見たものとソフトウェア的観点から見たものが存在

- 主な構成要素
  - クラス
  - 関連



←リバーシゲームに登場する物(の型)

(先手オブジェクト、後手オブジェクト)

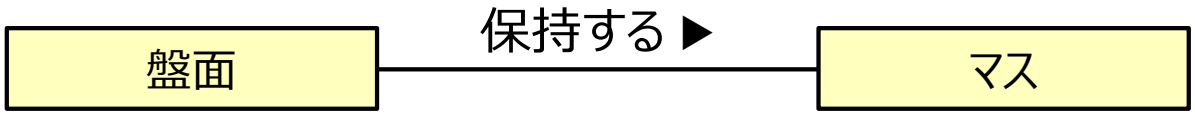
クラス図

■ 関連の分析

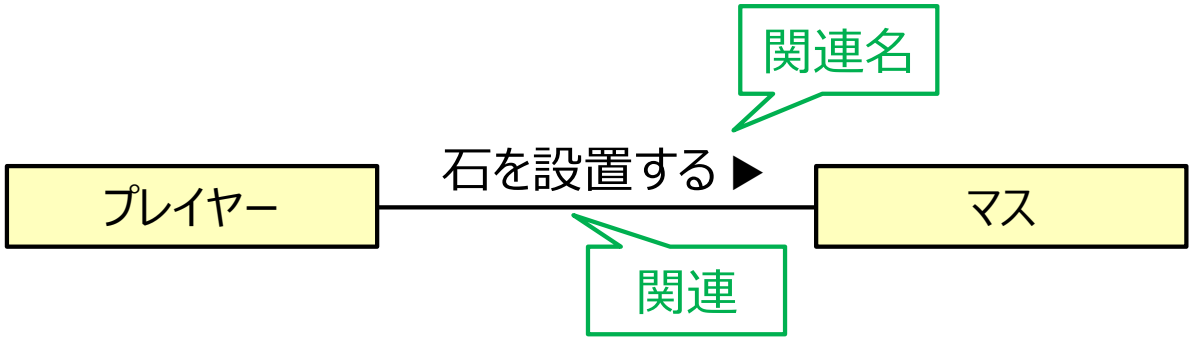
システム全体の概要の把握と共有
実装なしでシステムの全体図を顧客へ説明できる

- 盤面は縦８マス×横８マスである

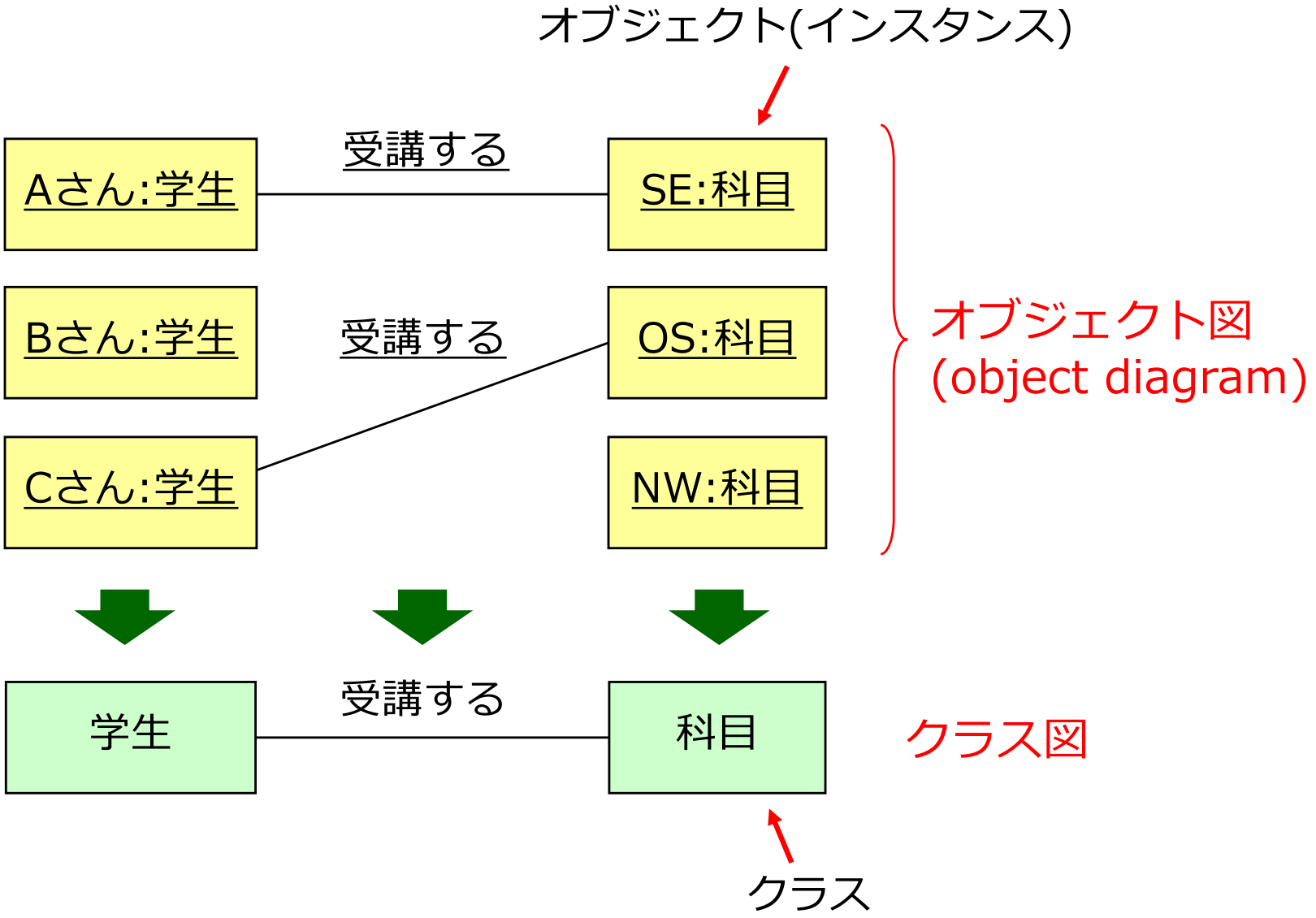
⇒ 盤面は64個のマスを保持する



- プレイヤーは自分の手番に、盤面の１つの空きマスに自色の石を１つ設置する

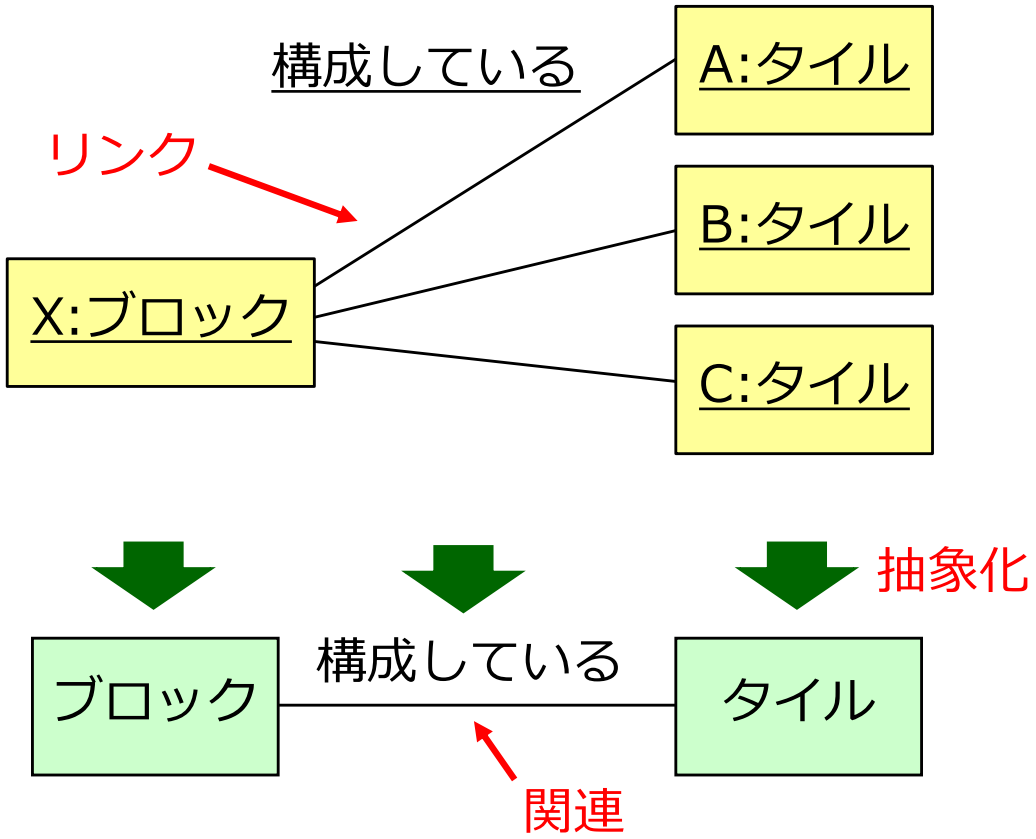


関連とリンク(補足)


オブジェクト(インスタンス)
Aさん:学生
受講する
SE:科目
Bさん:学生
受講する
OS:科目
Cさん:学生
NW:科目
オブジェクト図
(object diagram)
学生
受講する
科目
クラス図
クラス

関連とリンク

- リンク(link):
  - インスタンスどうしの一時的な協調関係
- 関連(association)
  - インスタンス間に成立する一時的な関係を抽象化したもの

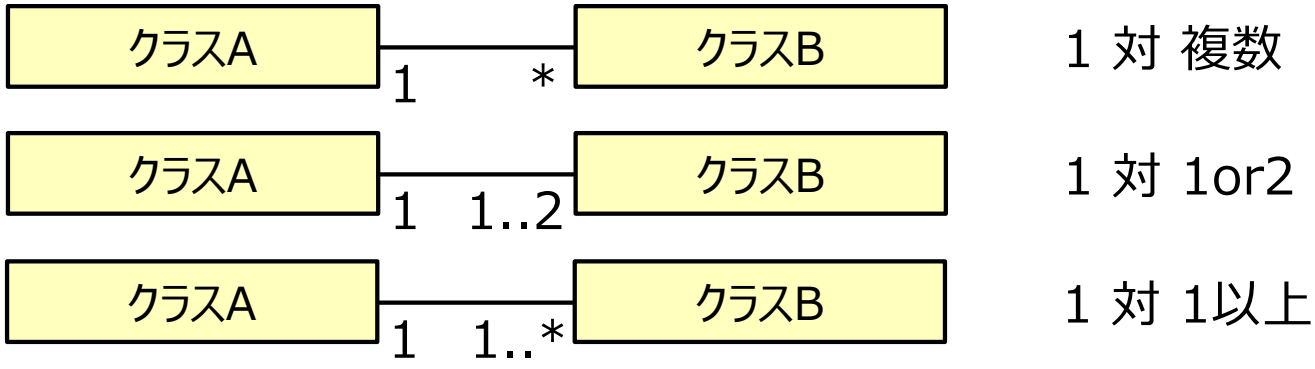


多重度(multiplicity)

- ある関連に関するインスタンスの数を明示

盤面は縦８マス×横８マスである
⇒ 盤面は64個のマスを保持する

何本リンクを引くことができるか

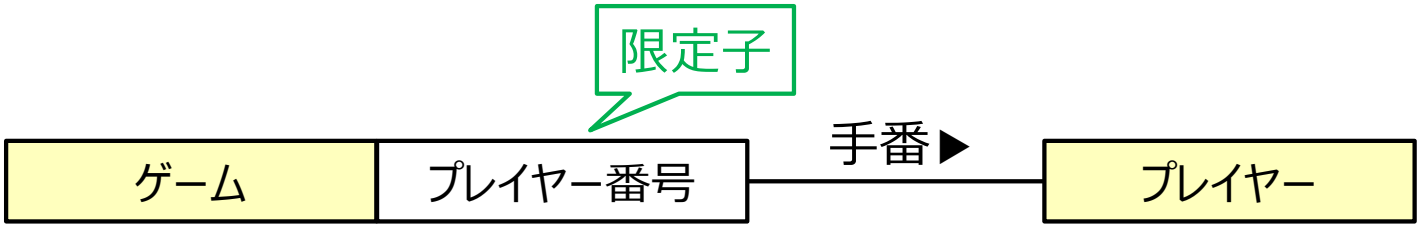


下限値..上限値

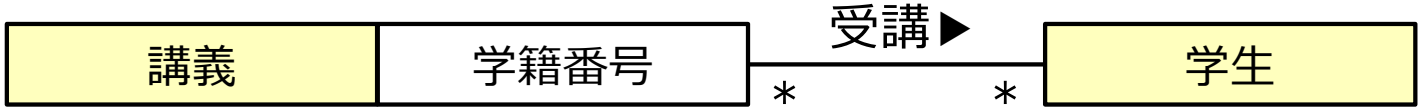
限定子(qualifier)

- 関連の相手のどの属性で参照するか
  - 関連の意味づけを深める

相手のオブジェクトを参照するために
必要な情報を明示



ゲームでプレイヤーを特定するにはプレイヤー番号が必要

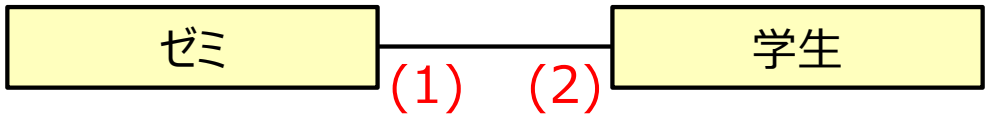


講義で学生を特定するには学籍番号が必要

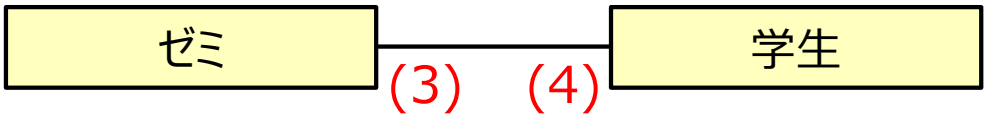
確認問題

■ 図が以下の意味になるように空欄を埋めよ。

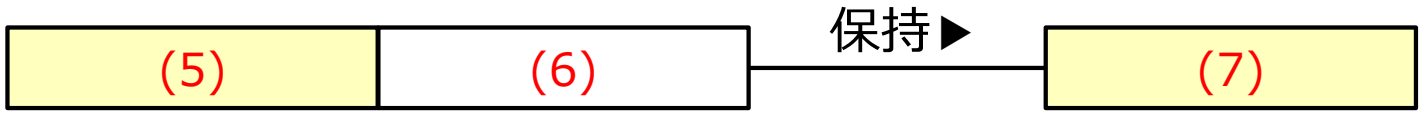
- 1つのゼミに複数の学生が所属する



- 1人の学生は複数のゼミに属し、
  1つのゼミには複数の学生が属する



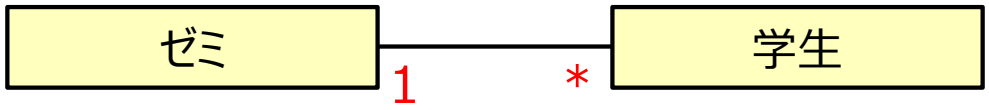
- ゼミクラスの学生番号という属性によって
  学生が参照される



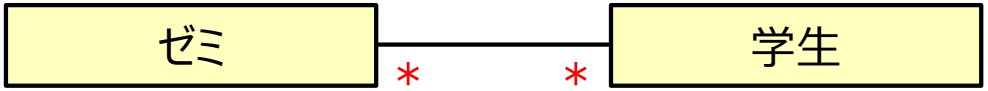
確認問題

■ 図が以下の意味になるように空欄を埋めよ。

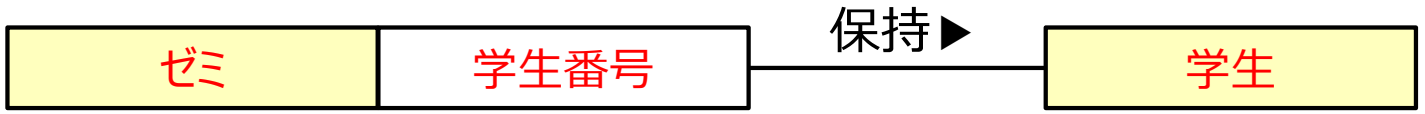
- 1つのゼミに複数の学生が所属する



- 1人の学生は複数のゼミに属し、
  1つのゼミには複数の学生が属する



- ゼミクラスの学生番号という属性によって
  学生が参照される



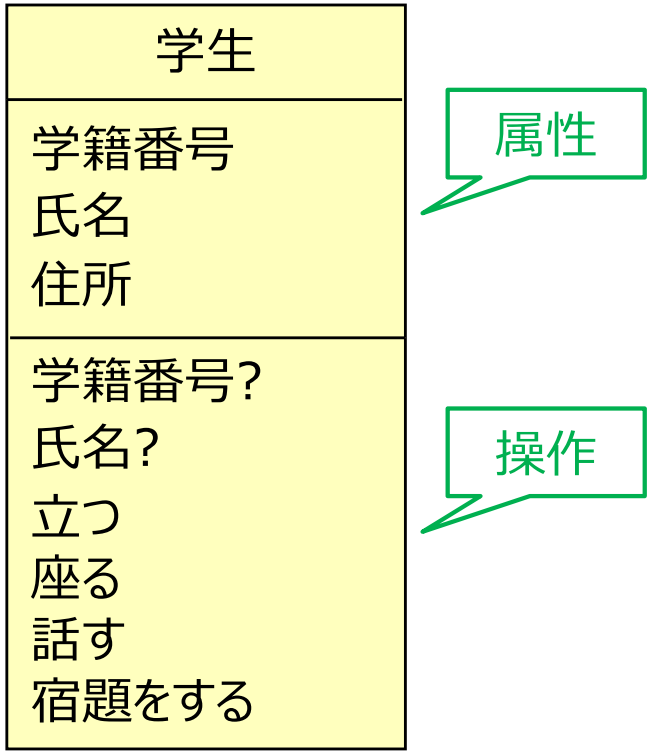
属性と操作

■操作の抽出

- 要求仕様中の動詞に対応することが多い

■属性の抽出

- 操作の対象となるデータ
- 操作に影響を与えるデータ



※クラス図、オブジェクト図ともにこの3区画

(例) 操作の抽出

※これまでの分析に基づき、元の要求文を変更

- ゲームは先手(プレイヤー)から開始する
- 手番を先手・後手に設定する
- プレイヤーは盤面に石を置く
- 挟まれた石すべてを自色に変える(ひっくり返す)

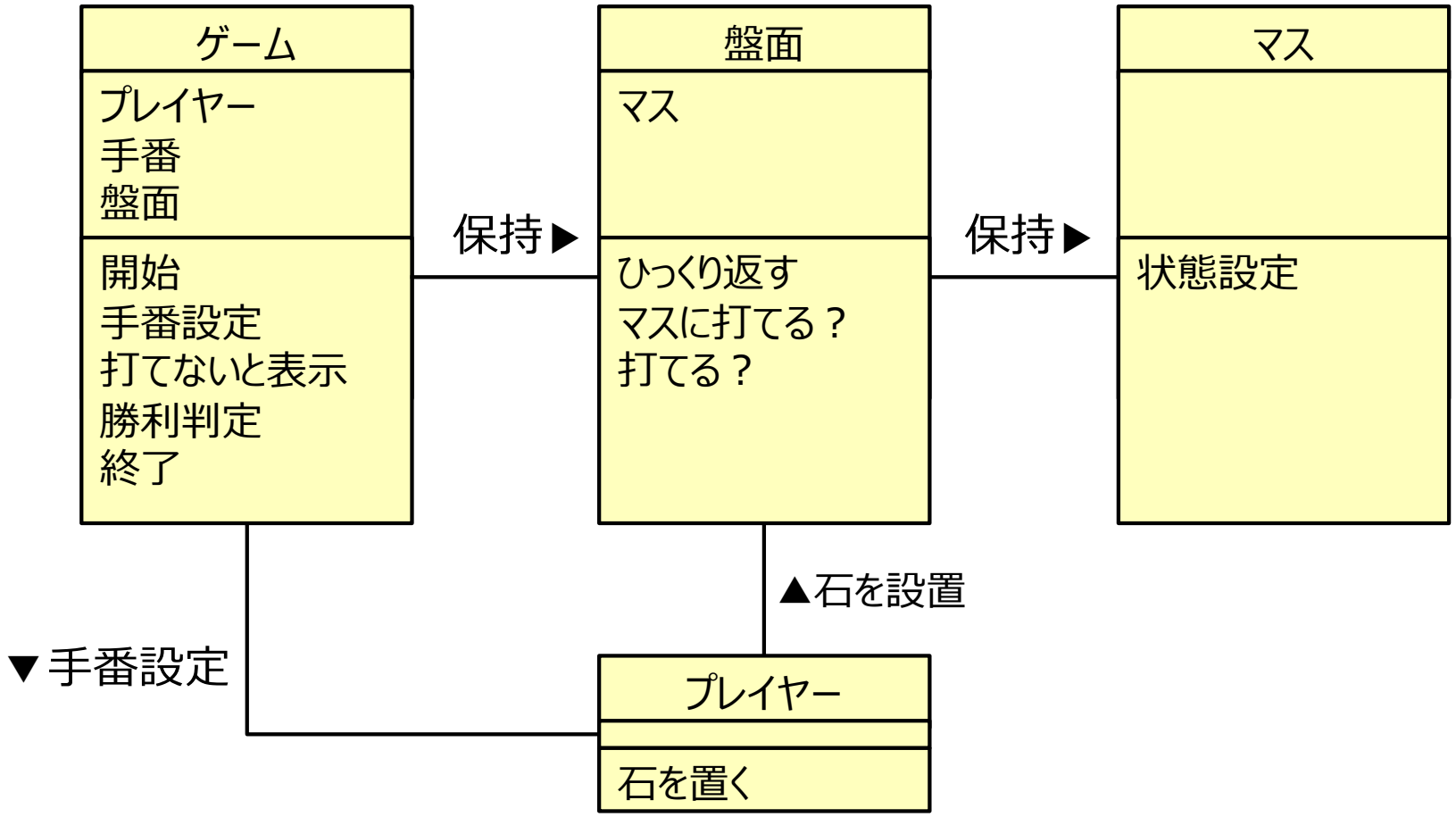
ゲーム	盤面	マス	プレイヤー
開始 手番設定 打てないと表示 勝利判定 終了	ひっくり返す マスに打てる？ 打てる？	状態設定	石を置く

(例) 属性の抽出

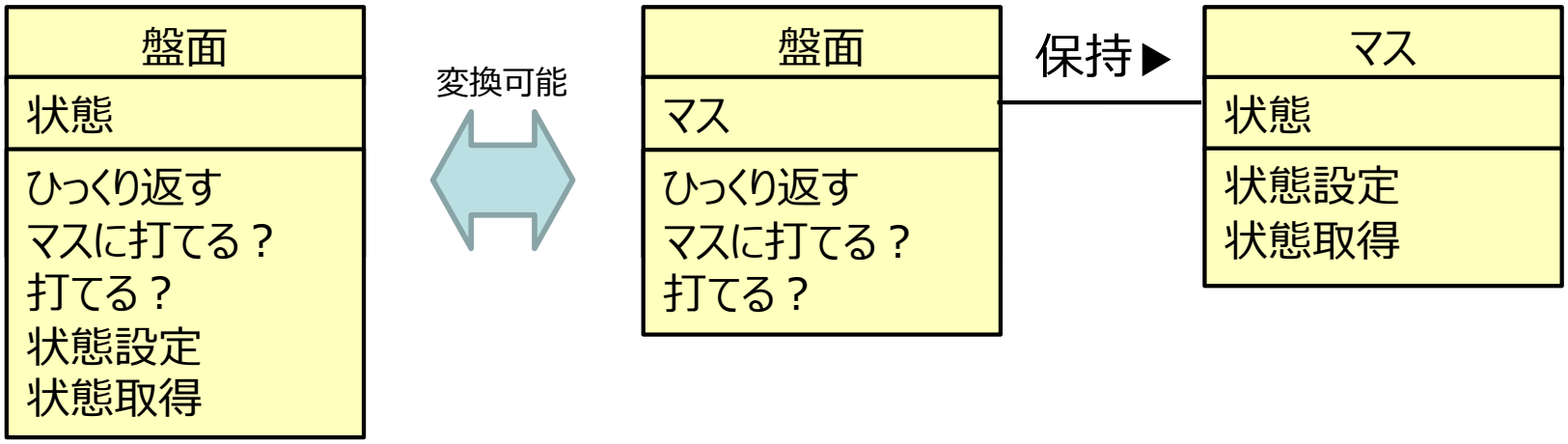
ゲーム	盤面	マス	プレイヤー
プレイヤー 手番 盤面	マス <u>縦マス数</u> <u>横マス数</u>		
開始 手番設定 打てないと表示 勝利判定 終了	ひっくり返す マスに打てる？ 打てる？	状態設定	石を置く

・静的な属性と操作
インスタンスではなく、クラスに属する属性や操作。
インスタンスを生成しなくても利用可能。
インスタンスごとに異なる属性値にはならない。
<u>下線</u>を引いて表現。
(例) 盤面のマスの数(縦8、横8)

(例) 関連・操作・属性


ゲーム
プレイヤー
手番
盤面
開始
手番設定
打てないと表示
勝利判定
終了
保持▶
盤面
マス
ひっくり返す
マスに打てる？
打てる？
保持▶
マス
状態設定
▲石を設置
▼手番設定
プレイヤー
石を置く

関連と属性



マスの状態も盤面の
属性として保持

別クラスとして切り出し

■ 判断基準

- そのクラスの本質的な特徴か？
- クラスとして扱うほど重要か？

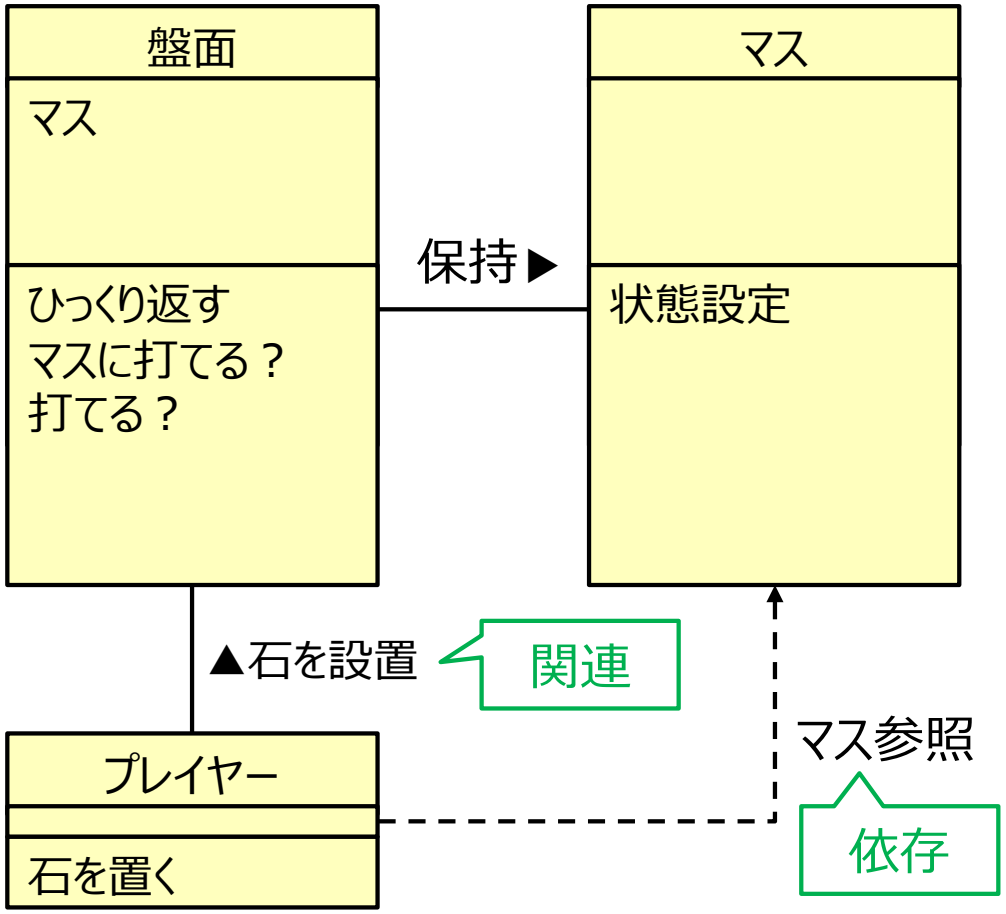
依存(dependency)

- クラスのインスタンス同士が一時的に協調する可能性を表現
  - 関連より弱い関係

関連：
　メンバ変数にオブジェクトを持つ
依存：
　一時オブジェクトとしてnewする

盤面はマスの属性を持つ
プレイヤーは盤面を持つ
　(盤面の状態が紐付く)

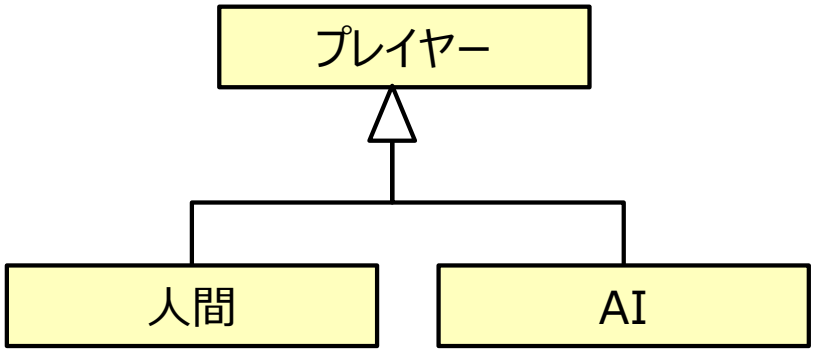
プレイヤーは石を置く際に
マスを参照し石が置けるか判断する



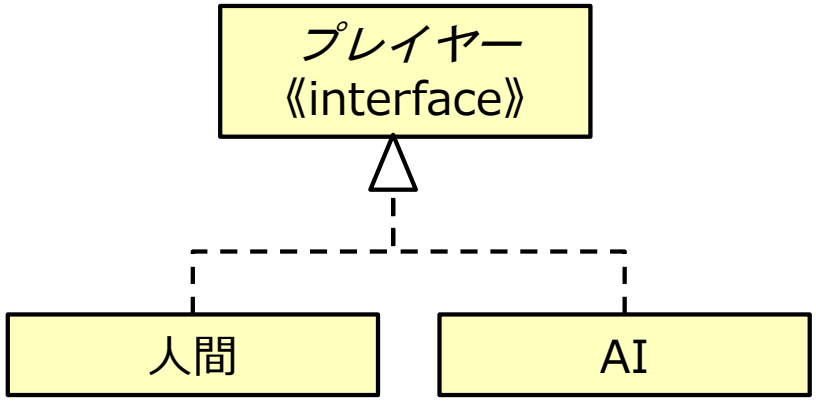
マス指定の際に一時的に参照する等

汎化(generalization)

■一方のクラスが他方のクラスを具体化している場合の関係



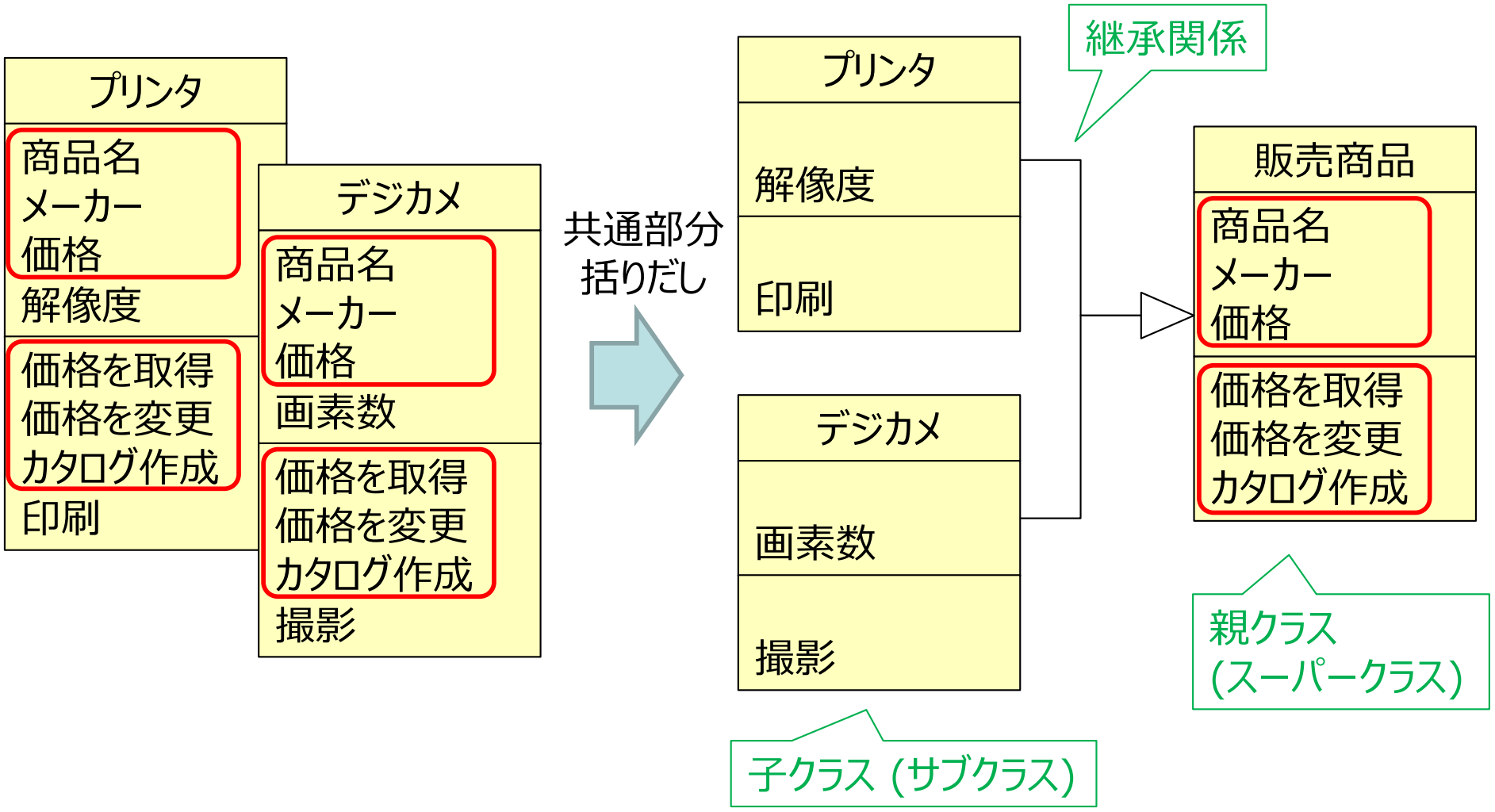
継承
(人間とAIはプレイヤーを継承する)



インタフェース継承
(プレイヤーが内部実装を持たないinterfaceの場合)

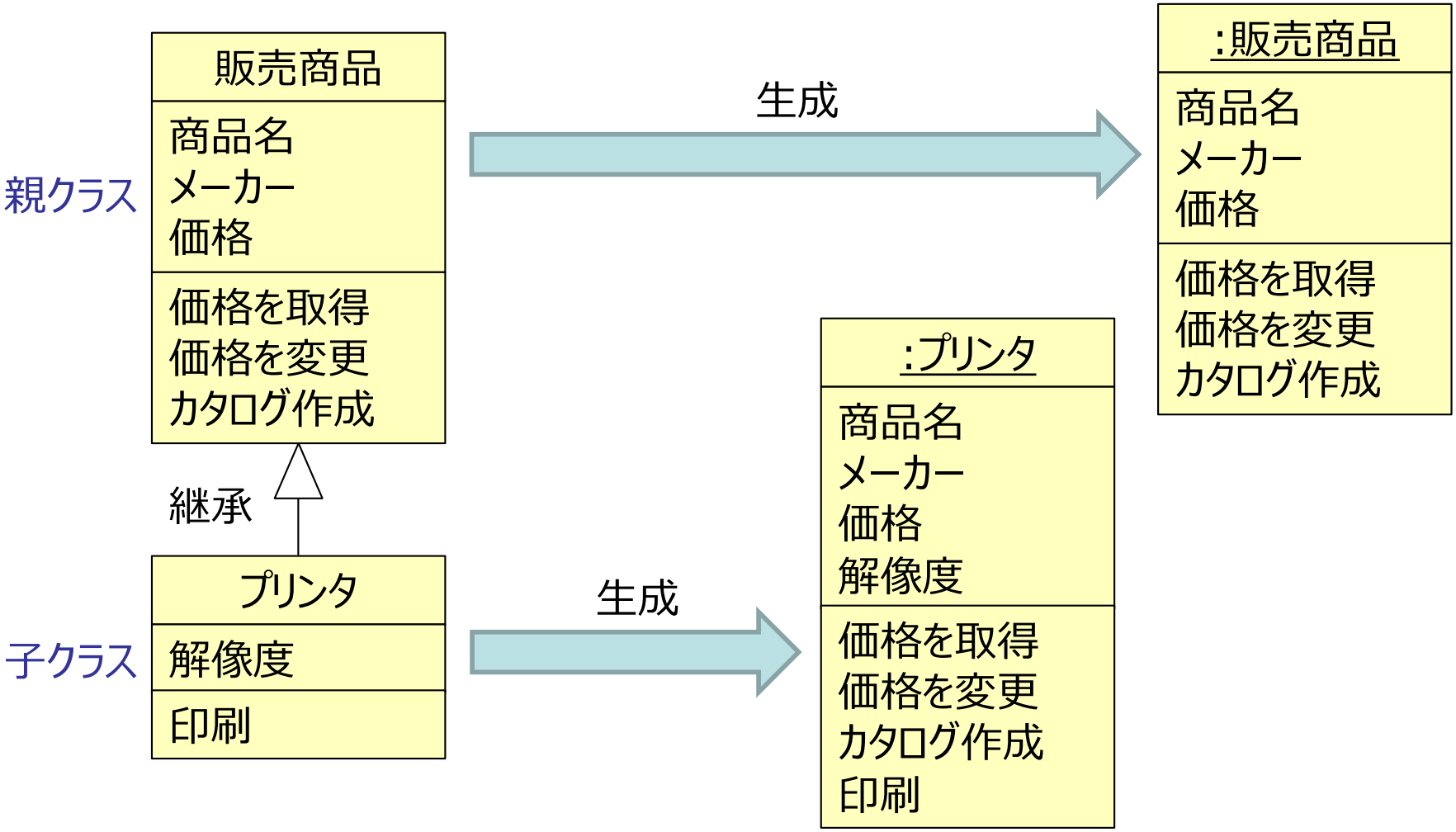
継承(inheritance)

■ 複数のクラスにおける共通の性質を共有させる仕組み

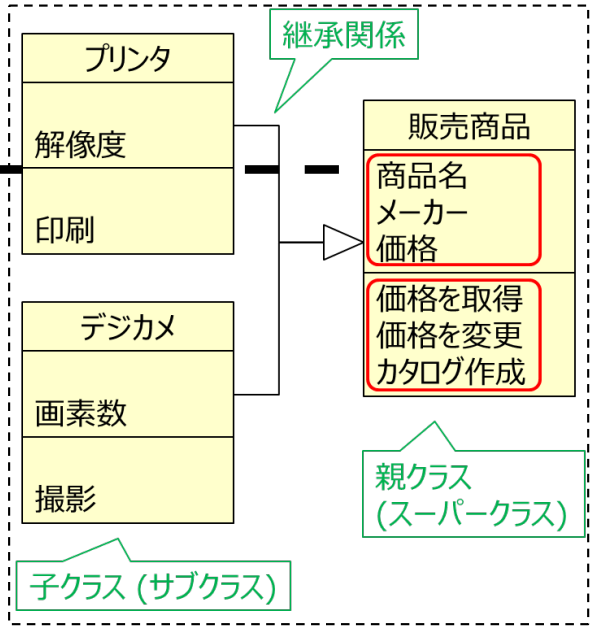


継承とオブジェクトの生成

■ 子クラスのインスタンスを生成すると、
親クラスのメンバも含まれたものになる



継承



■ is-a関係

- 親クラスと子クラスの関係
  例：「デジカメは販売商品である」
  Digital camera is a sales product.

■ 継承を用いる利点

- 再利用性：同じデータ構造や機能等を再利用
- 拡張性：元のクラスをベースに新しいデータや機能を拡張しやすい

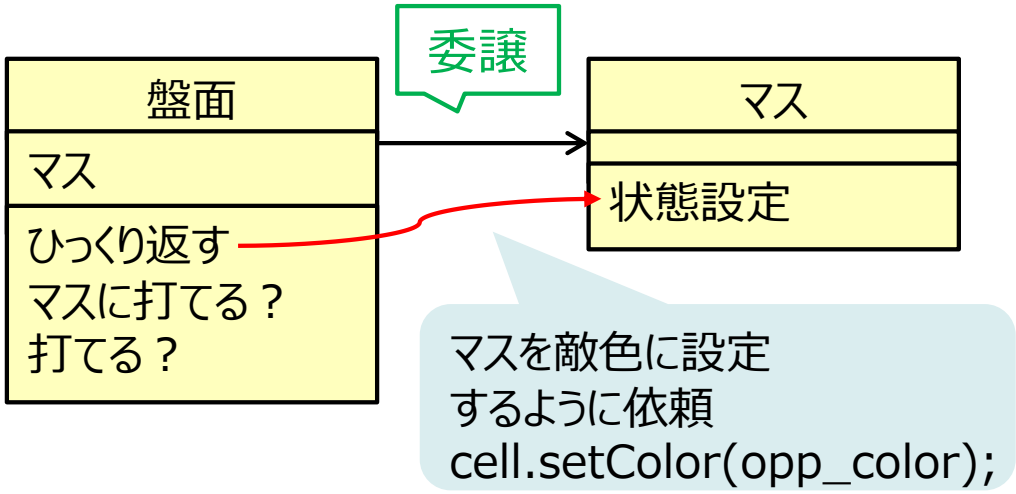
■ 継承を用いる上での欠点

- 親クラスの変更が子クラス全てに影響を与えるため、子クラスでバグが生じる可能性

委譲(delegation)

■ 別のインスタンスに**処理の一部**を依頼すること

継承は**処理の全て**を引き継ぐ



部分的な処理を他のメソッドに委ねることができる

- 呼び出し側では、インタフェース(どのように呼び出すか)さえ知っておけば良い
  - 実際の呼び出し先を後から設定する仕組みもある
    →動的束縛

継承と委譲(Java)

継承

```
class Animal {
 void cry() {
  System.out.println(”make a sound");
 }
}
class Dog extends Animal {
 void cry() {
  System.out.println(”bark");
 }
}
class Bird extends Animal {
 void cry() {
  System.out.println(”tweet");
 }
}
```

委譲

```
class Message{
 void write(String text){ ... }
 boolean delete(int target){ ... }
 boolean send(String text){ ... }
}
class Mail{
 Message message;
 String title;
 String body;
 UserRepository(String title,
                        String body){
 message = new Message();
 this.title = title;
 this.body = body;
}
 boolean send(int userId) {
 return this.message.send();
 }
}
```

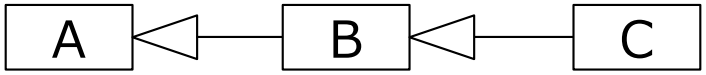
確認問題

■ 以下の各文は正しいか。○か×で答えよ。

- 名詞抽出法では、要求仕様中の名詞がメソッドに対応すると考える。
- 名詞抽出法で抽出された名詞はすべてクラスやオブジェクトとして実装すべきである。
- オブジェクト図では、1つのクラスから生成されたインスタンスは1つだけ描画される。
- 静的な属性は、生成するオブジェクトごとに異なる値を取り得る。
- クラス図において、内部実装を持たないインタフェースの名前には下線を付与する。
- 継承を用いる利点として、同じデータ構造や機能の再利用が挙げられる。
- 親クラスの機能を拡張することを目的に継承を用いるべきではない。

確認問題

- is-a関係に相当するのは次のうちどれか。
  継承、委譲、集約
- インスタンス同士が一時的に協調する可能性を
  表現するものはどれか。
  継承、複合、依存
- 正しい方を選べ。
  - 親クラスは子クラスの(汎化・特化)である。
- クラスA～Cは下図のような継承関係にあり、
  クラスAのメンバはa,b,c、クラスBのメンバはd,e、クラスCの
  メンバはfとする。このとき、A,B,Cのインスタンスが
  メンバとして保持するものをそれぞれ答えよ。
  (ただし、すべてのメンバは継承関係にあるクラスに対して
  公開されるものとする。)

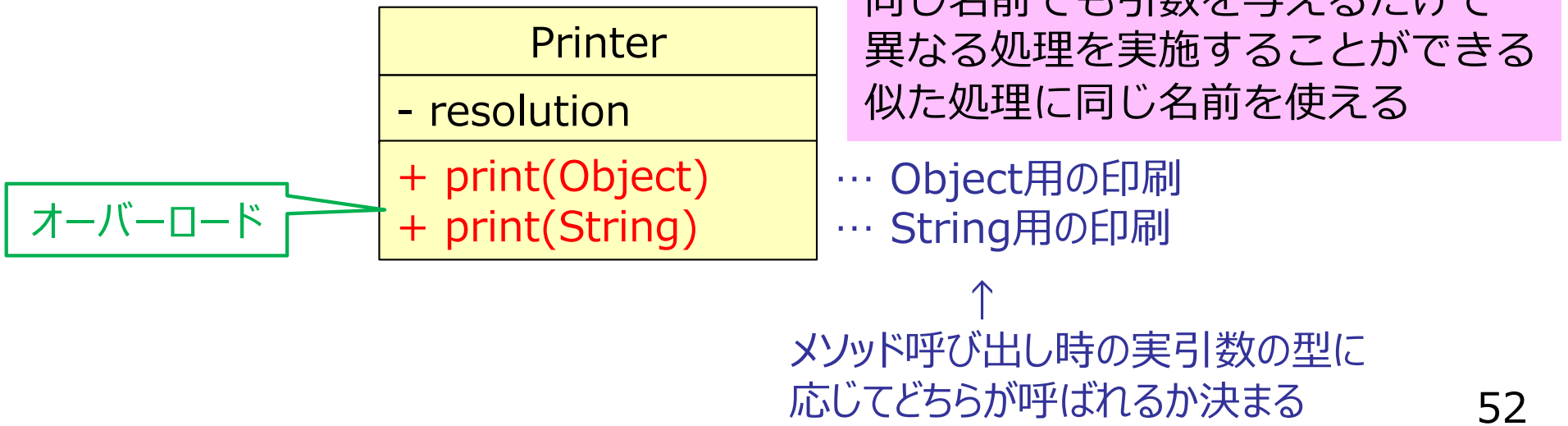


講義内容

- 静的モデリング
  - オーバーロード
  - オーバーライド
  - 抽象メソッド
  - 抽象クラス、インタフェース
  - 多態性
  - 動的束縛

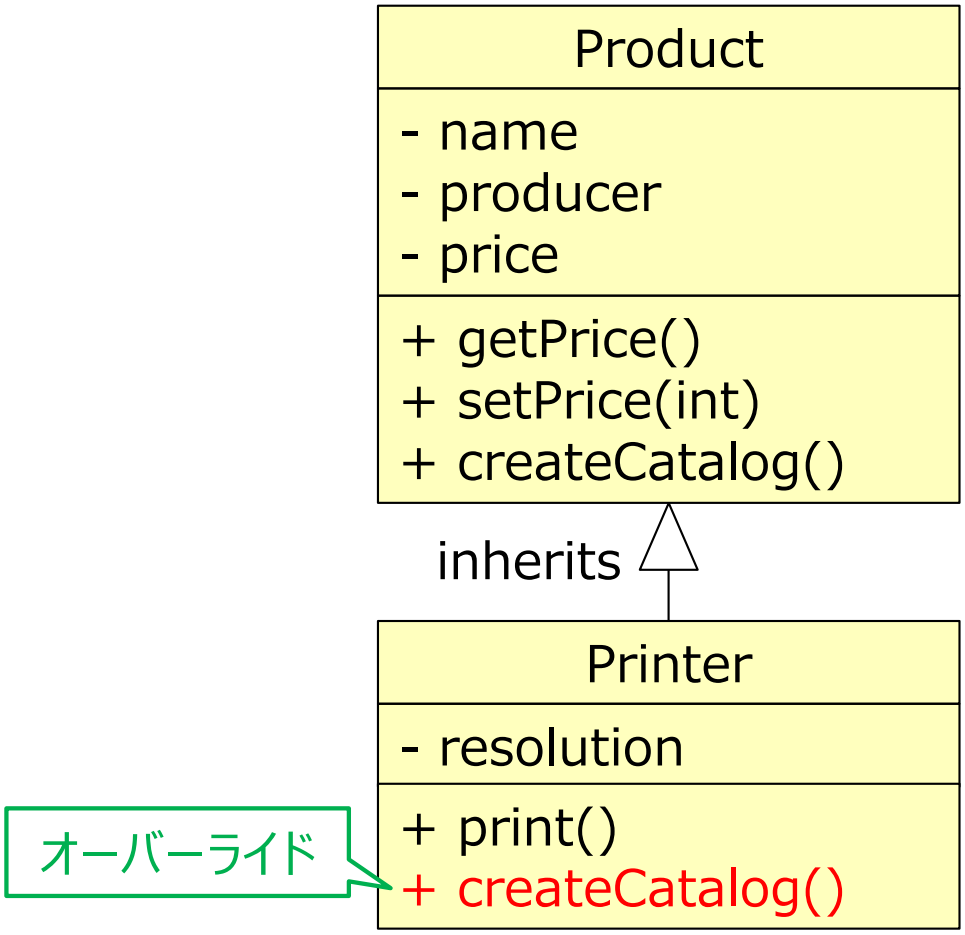
オーバーロード(overload)

- 同じ名前のメソッドを複数宣言すること
  (引数の数や型が互いに異なる)
  - Javaの場合、引数が異なれば
    同じ名前のメソッドを複数宣言可能
  - 名前と引数が同じで戻り値型のみ異なる
    メソッドは複数宣言**できない**



オーバーライド(override)

■ 子クラスのメンバが親クラスのメンバを再定義(上書き)すること



メソッド名はそのまま
自クラスに適した内容に変えられる

Printerインスタンスに対して
createCatalogメソッドを呼び出すと、
Printerクラス内で定義された
createCatalogが実行される

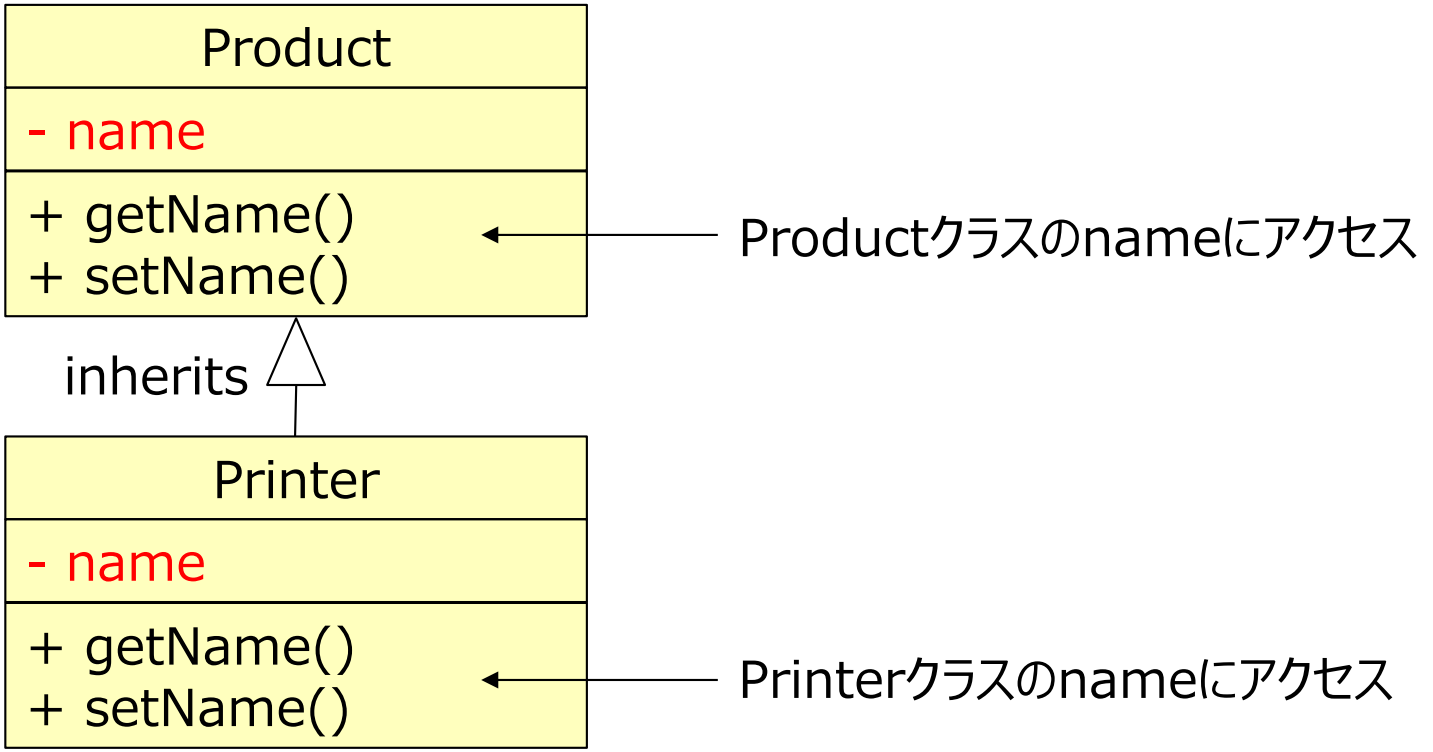
… 親(先祖)クラスと同じシグネチャ
(名前も引数も戻り値型も同じ)

注意点

- 意図しないオーバーロード・オーバーライドに注意
  - (例) P.m(P) と C.m(C) がある場合
    c.m(p), c.m(c)の呼び出し対象は…？

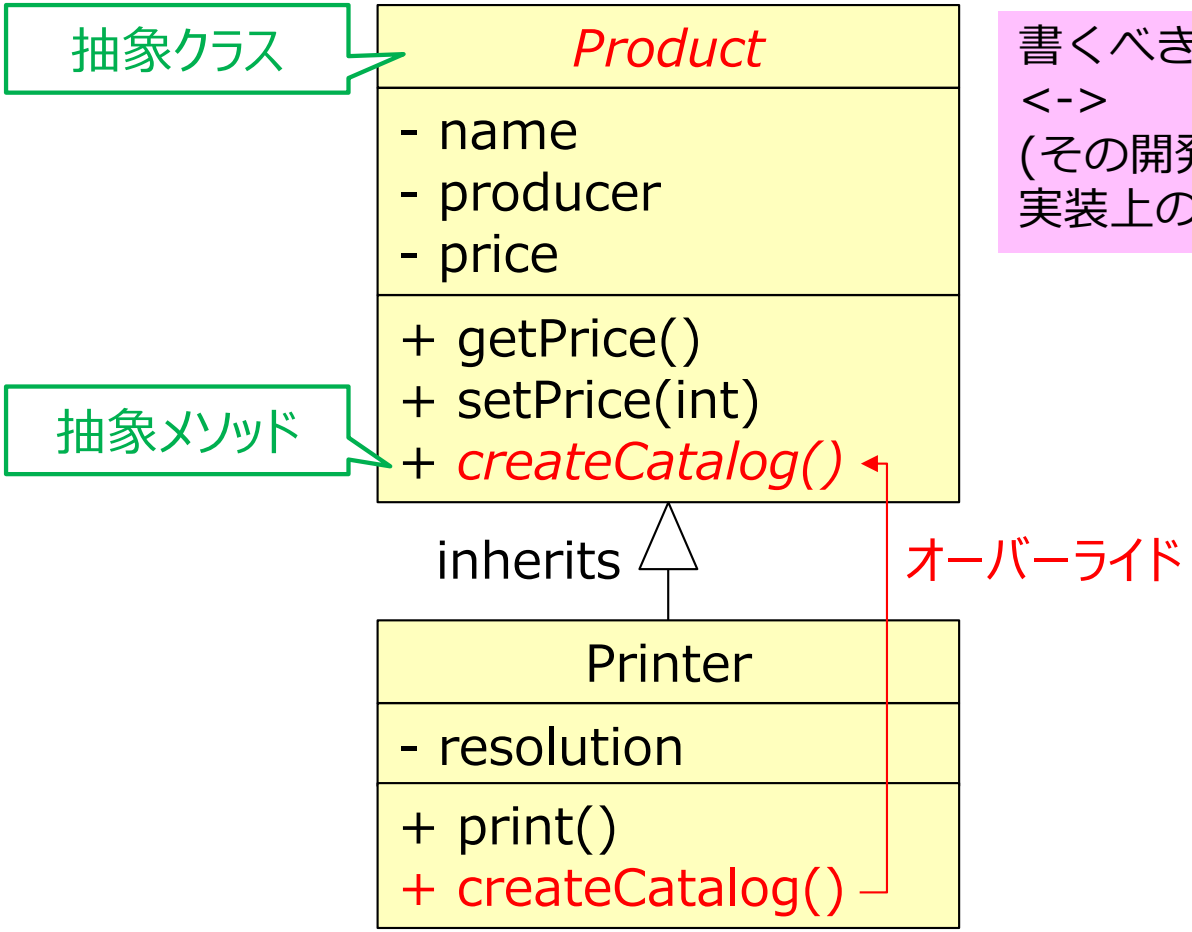
C extends P
P p; C c;

- フィールドはオーバーライドできない (隠蔽される)
  → (実用上)そのような実装はしてはいけない



抽象メソッド、抽象クラス

- 抽象メソッド (abstract method)
  - 中身(実装)がないメソッド
  - シグネチャのみ決定
    (中身は子・子孫クラスが決める)
- 抽象クラス (abstract class)
  - 抽象メソッドを持つクラス
  - インスタンス生成できない



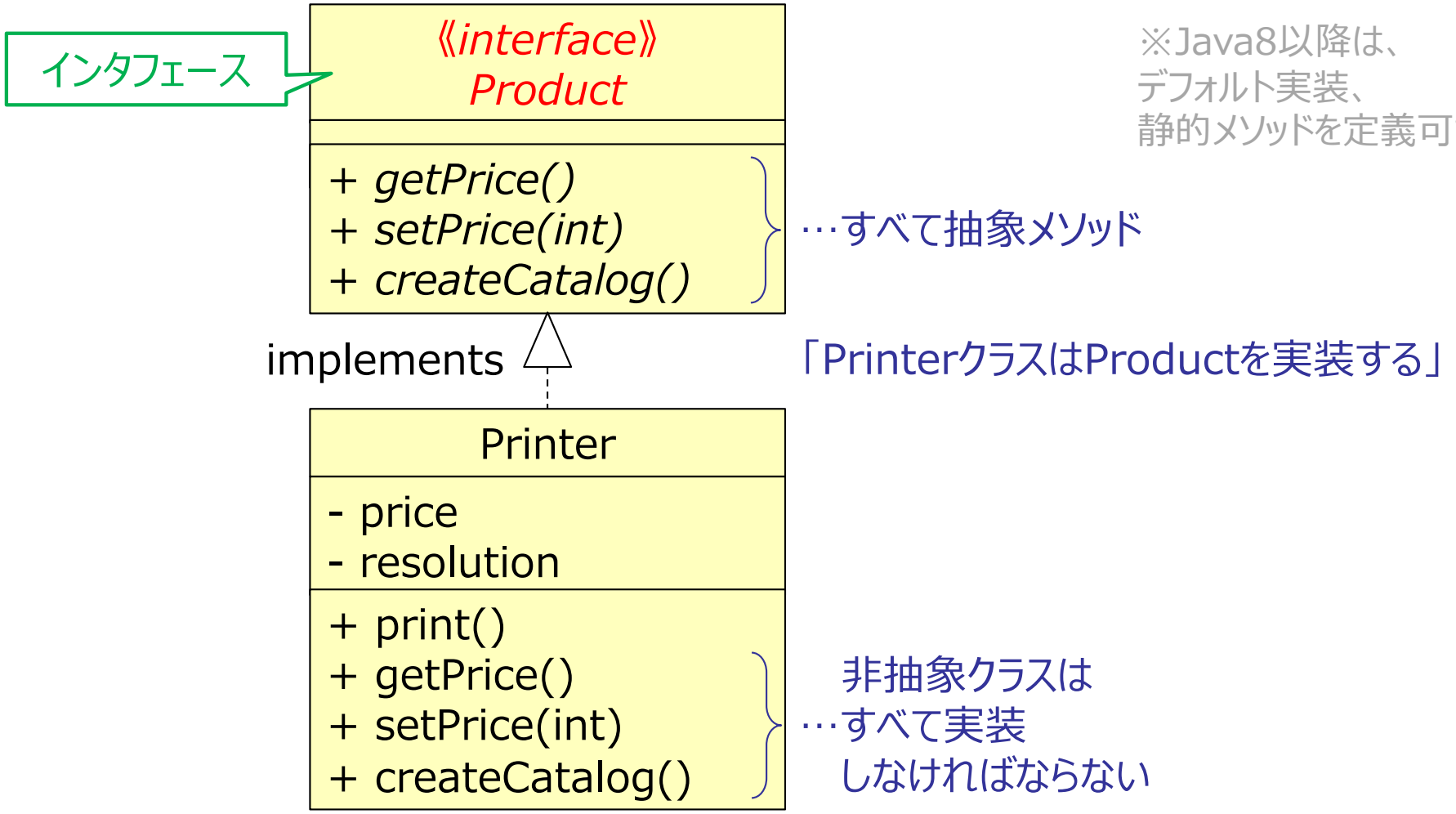
書くべきメソッドが明示される
<->
(その開発における)
実装上のルールが明示される

インタフェース(interface)

■ メソッドのシグネチャを定義

※Javaのinterfaceでは
フィールドを宣言できるが、
すべて定数(public static final)
として扱われる

※Java8以降は、
デフォルト実装、
静的メソッドを定義可能



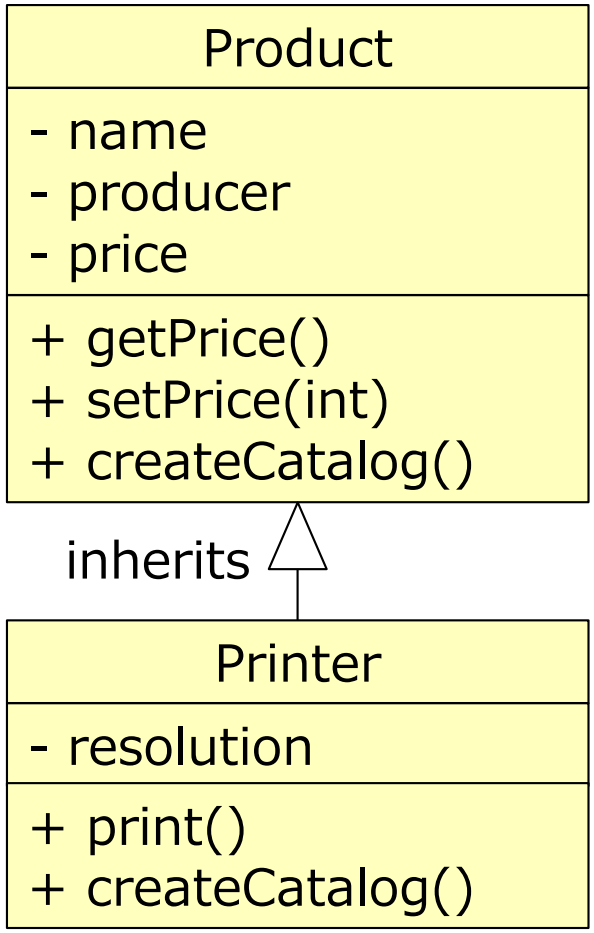
抽象メソッドやインタフェースの利点

- 実装とインタフェースの切り分け
  - 各クラスの責任を明示
    (あるインタフェースを実装する＝
    そこに含まれる全てのメソッドを実装する責任)
  - 実装内容は変わったとしても、
    インタフェース(責任)は変える必要がない
    (直接実装メソッドを参照するよりも疎結合)
- 子クラスに実装を強要する
  - 子クラスで何を作らなければならないか
    を明示できる　その開発における実装上のルールが、コード上で明示される
  - 親クラスで処理の流れのみ決めておき、
    その中身は子クラスに任せることが可能

ポリモーフィズム

多態性(polymorphism, 多相性)

- 1つのインスタンスが複数のクラスの
  インスタンスとみなせる性質

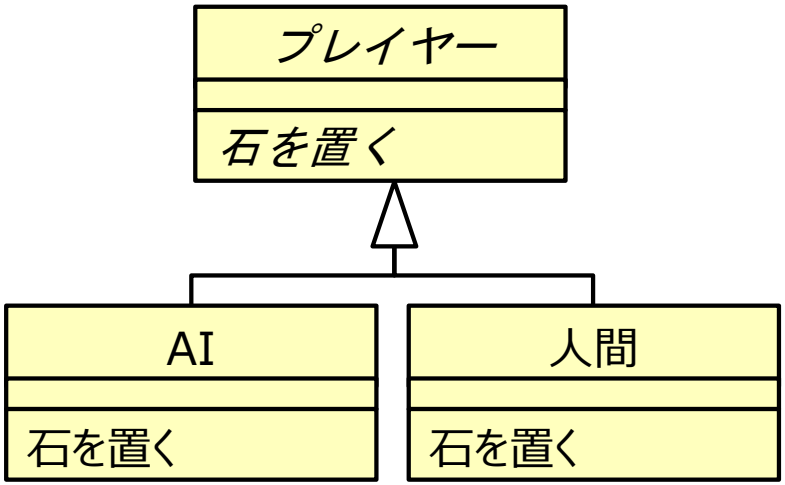


Printerインスタンスは
Productインスタンスとみなすこともできる

Product型変数には
Printer型もProduct型も代入できる

多態性の例

■ 複数のサブクラスを持つ場合
→ インスタンスの型による処理の変更が可能



AIインスタンスも人間インスタンスも
プレイヤー型とみなせる

```
AI ai;
Human human;
Player current = ai; //現在どちらの手番か

...

for(;;){
  current = (current == human ? ai : human);
  current.putStone();
  ...
}
```

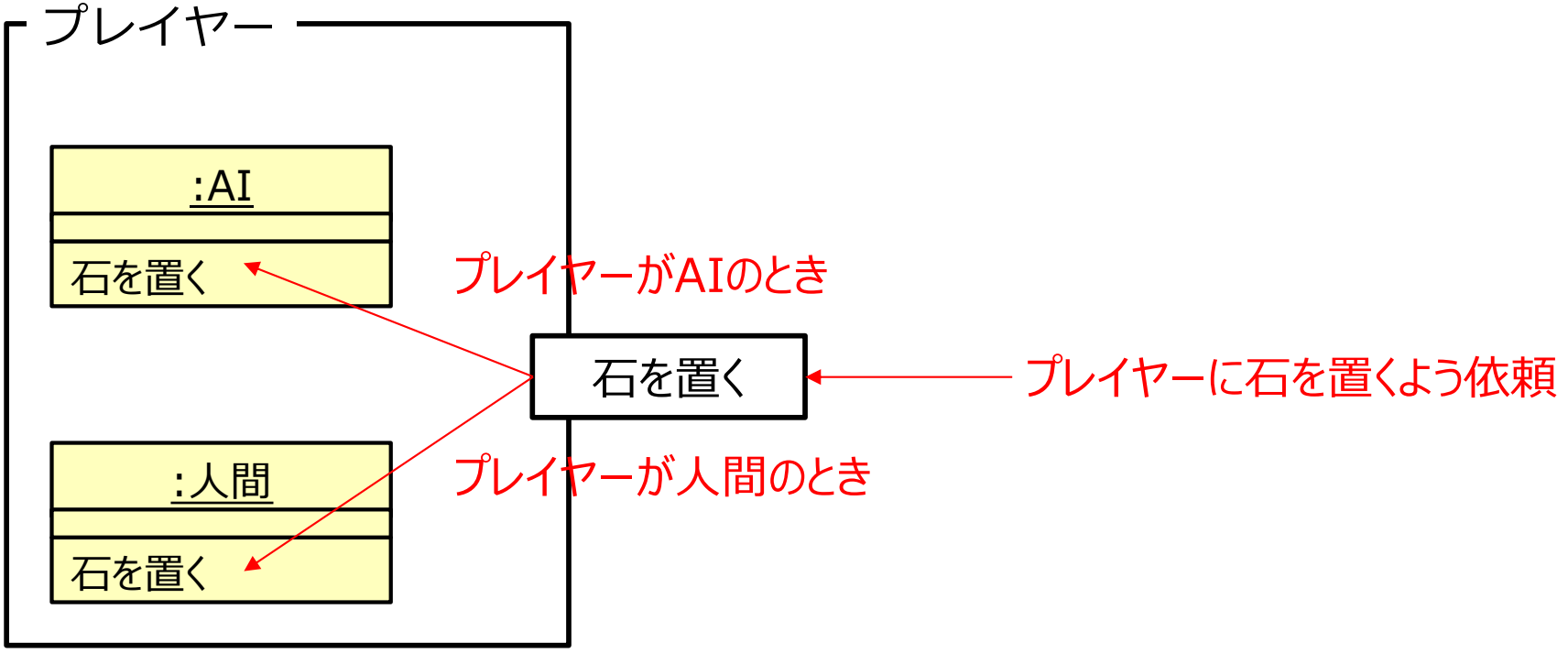
currentにはaiも
humanも代入可

currentがaiならAIクラスのputStoneを実行
currentがhumanならHumanクラスのputStoneを実行

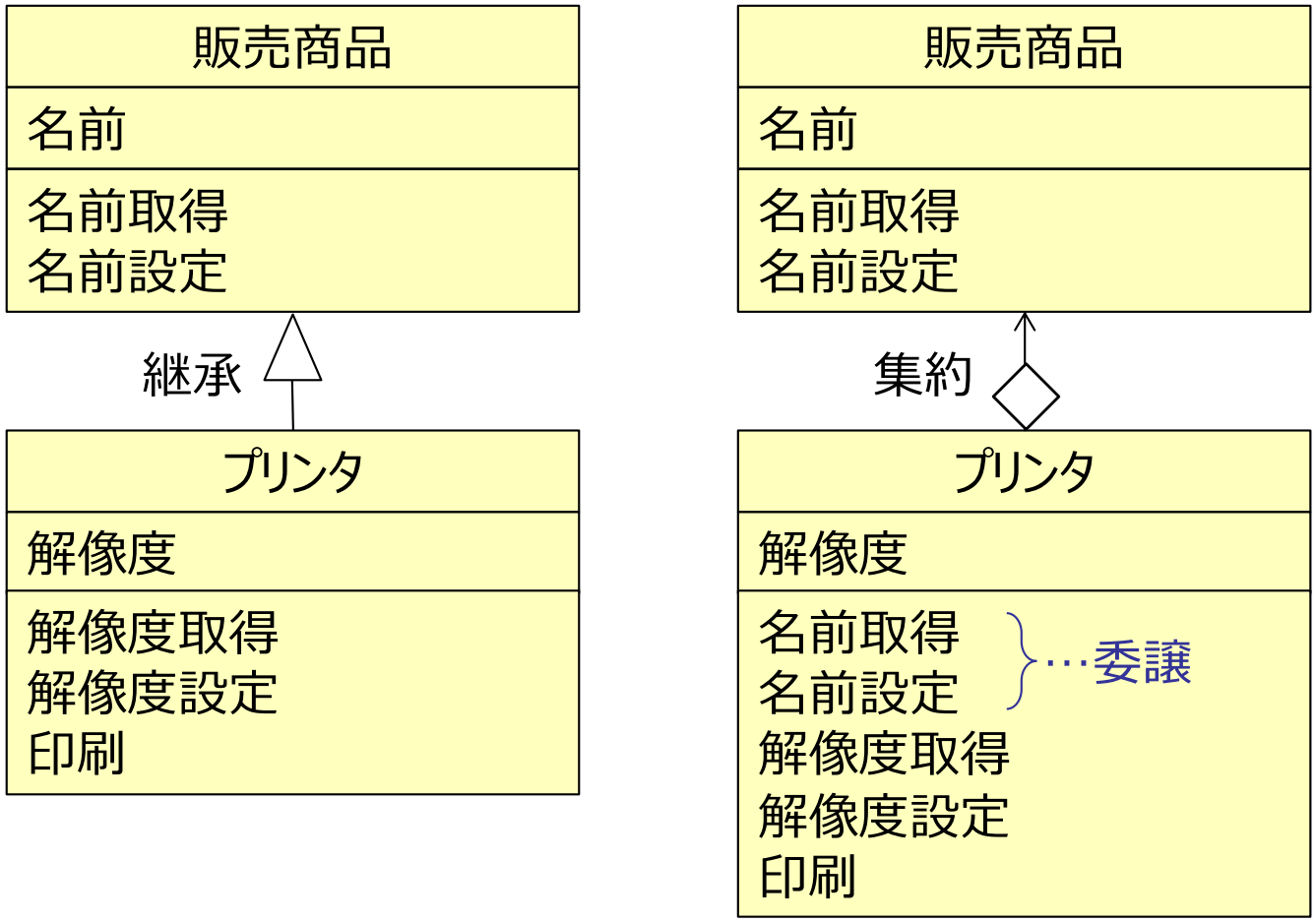
動的束縛 (dynamic binding)

■ 操作が呼び出される度に、操作の対象の型が適切に選択されること

呼び出し側は、呼び出し対象の実際の型を明示的に指定する必要がない



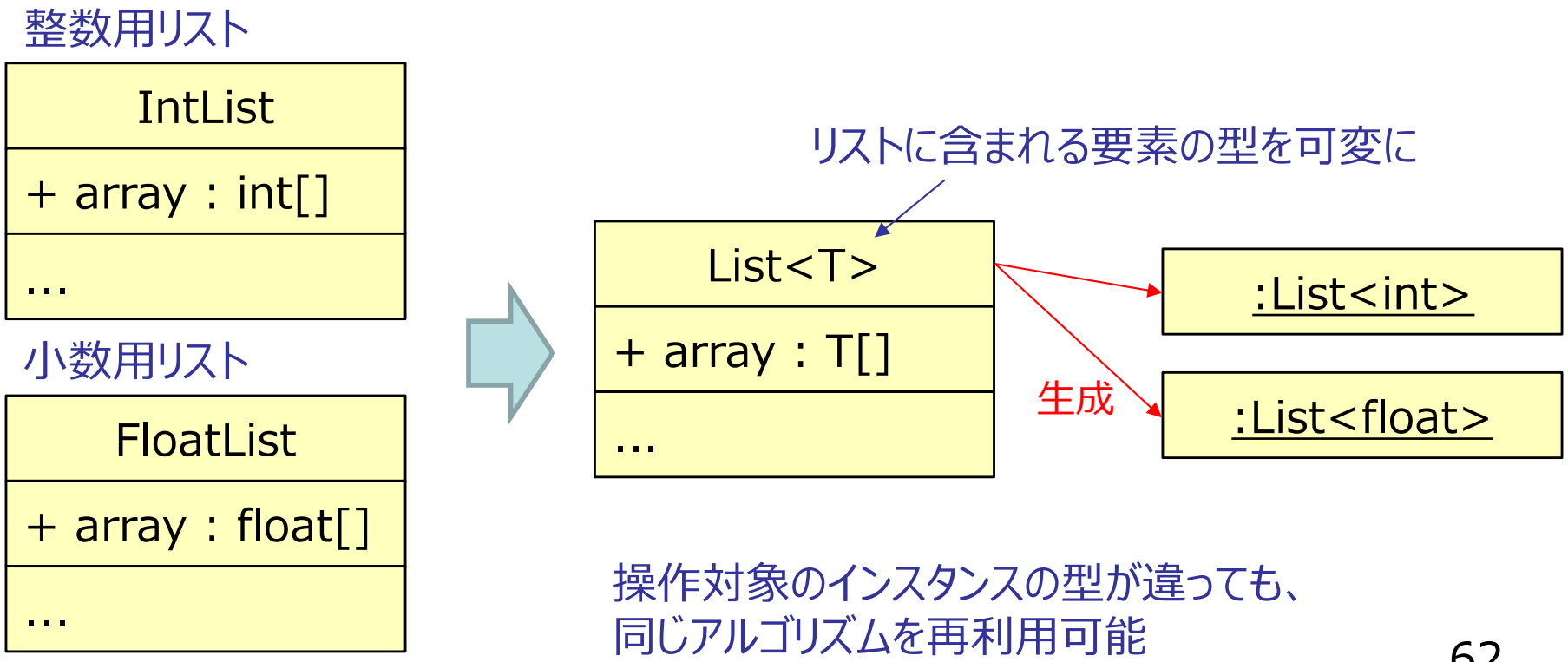
継承・集約によるクラスの機能拡張



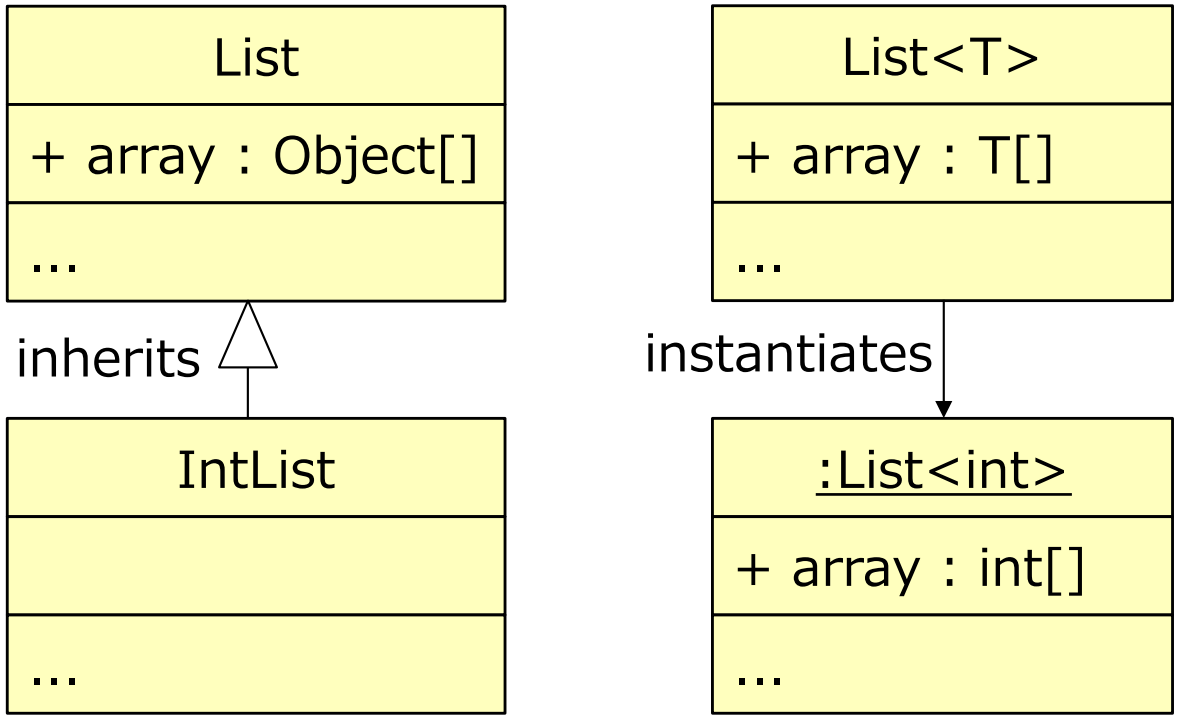
- どちらでもプリンタインスタンスから販売商品クラスの機能を使用可
- 集約の場合、
  - プリンタと販売商品は別クラスのインスタンス → オブジェクト間のリンクとして関係が出現
  - 集約の場合、プリンタを販売商品とみなすことはできない → 動的束縛不可

総称(generics)

- クラスの実装の一部で使われる型を可変にし、実行時に適切な実体を提供する仕組み
  - クラスやメソッドに「型引数」(type parameter)を指定



総称と継承



arrayにはアクセスできるが、
中身がintであることを
保証できない

子クラスで新しいフィールドや
メソッドを追加可能

arrayにアクセスでき、
中身がintであることも
保証できる

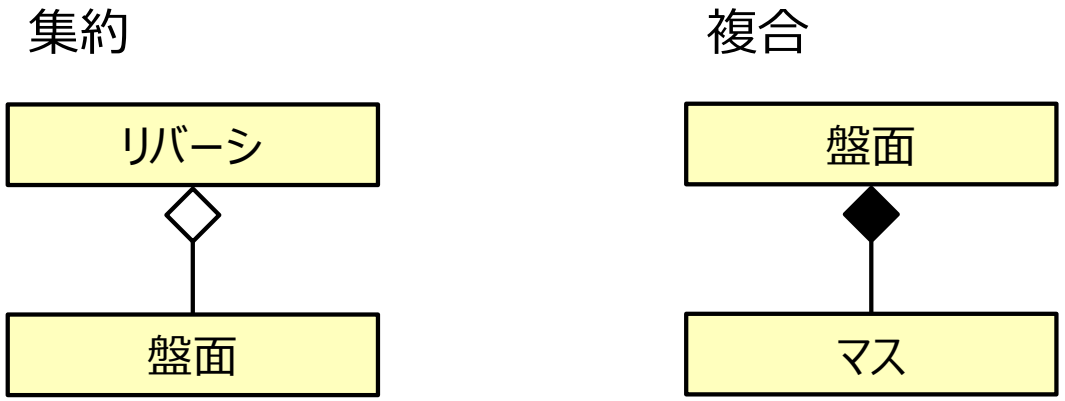
集約(aggregation)と複合(composition)

■ 集約　　<->is-a関係(継承)

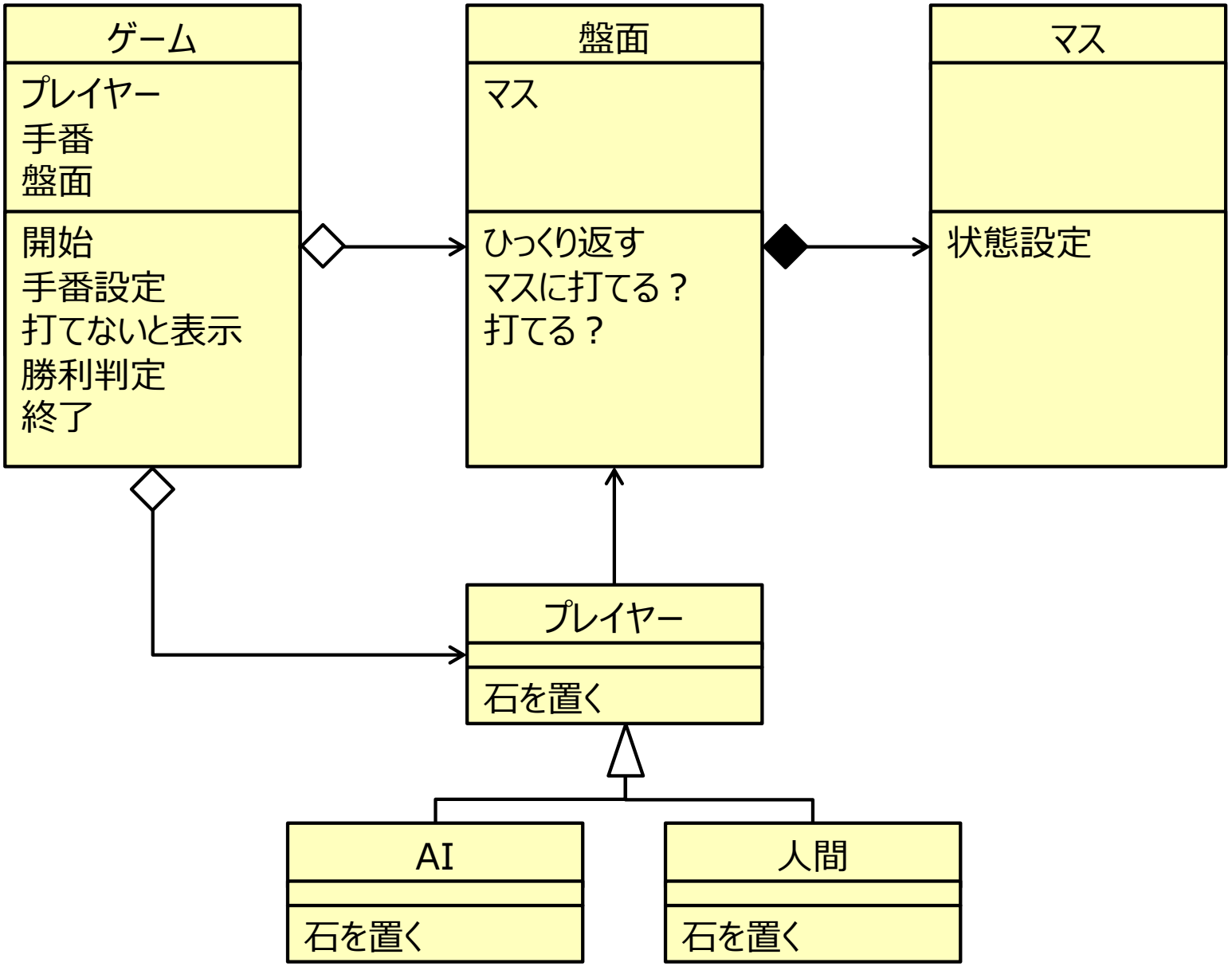
- has-a関係 (全体と部分の関係、所有関係を表現)
  例：「リバーシでは盤面を使用する」

■ 複合

- 集約の中でも特に強い関係を持つ
  (強い所有関係、同一の生存期間)
  例：「盤面はマスを包含する」



(例) 継承・集約・委譲


ゲーム
プレイヤー
手番
盤面
開始
手番設定
打てないと表示
勝利判定
終了
盤面
マス
ひっくり返す
マスに打てる？
打てる？
マス
状態設定
プレイヤー
石を置く
AI
石を置く
人間
石を置く

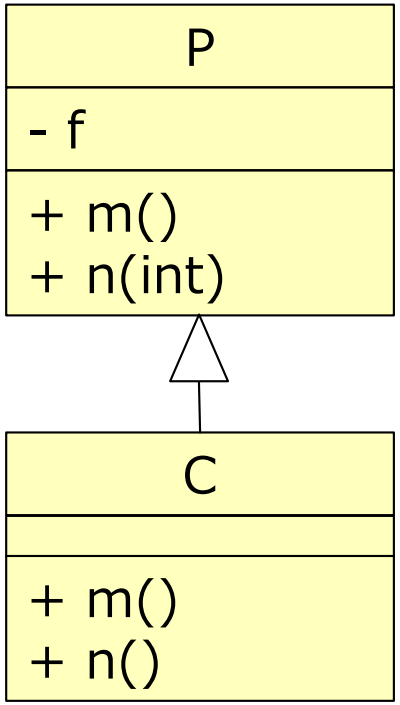
確認問題

- Javaプログラムにおいて、クラスCにメソッド"int m(int x)"がある。以下のそれぞれのメソッド宣言をクラスCにおいて行うと、次のどの結果になるか。

  (a)宣言可能かつオーバーロード関係になる
  (b)宣言可能かつオーバーロード関係にならない
  (c)宣言不可能

  - (1) void m()
  - (2) void m(int x)
  - (3) int m(int y)
  - (4) int n(int x)
  - (5) int m(int x, int y)
  - (6) int m(long x)

確認問題



- 以下のメソッド呼び出しで実際に実行されるメソッドの所属クラスを答えよ。
  - (1) P p = new P();
p.m();
  - (2) P p = new C();
p.m();
  - (3) C c = new C();
c.m();
  - (4) C c = new C();
c.n(1);
  - (5) C c = new C();
((P)c).m();

確認問題

■ 以下の各文は正しいか。○か×で答えよ。

- 抽象メソッドを持つクラスはインスタンス生成できない。
- インターフェースを実装するabstractでないクラスは、インターフェース内のメソッドすべての実装を持たなければならない。
- インタフェース導入の利点として、子クラスや子孫クラスにおいてその内容の実装をしなくてもよくなることが挙げられる。
- インタフェースを介してメソッド参照することで、直接参照するよりも疎結合になるという利点がある。

■ 以下の各文は次のうちどの言葉と最も関係があるか。

選択肢：多態性、動的束縛、総称

- クラス実装に含まれる型を可変にする。
- あるインスタンスが他のクラスのインスタンスともみなせる。
- 操作呼び出しごとに対象オブジェクトの型が選択される。