

## Javaプログラミング実習

# 22. 抽象クラスとインターフェース

株式会社ジードライブ

# 今回学ぶこと

---

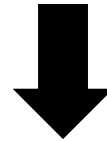
- 開発への取り組み方
- 継承を前提としたクラスの必要性和課題
  - メソッドの定義の課題
  - 多重継承の課題
  - 課題に対する解決策
- 抽象クラス
  - 抽象メソッド
- インターフェース
- 多態性

# 開発への取り組み方 1

---

事例：

サッカーゲームの開発に関わることになった



取り組み方 1：

- FieldPlayerクラスを作り、歩く、走る、ジャンプ、ドリブル、シュート、ヘディングの機能を実装した
- 同時にGoalKeeperクラスの作成も始め、歩く、走る、ジャンプ、パス、パンチング、キャッチの機能を実装した

# 開発への取り組み方 1

## FieldPlayer

```
walk() { 歩く }  
run() { 走る }  
jump() { ジャンプ }  
dribble() { ドリブル }  
shoot() { シュート }  
heading() { ヘディング }
```

## GoalKeeper

```
walk() { 歩く }  
dash() { 走る }  
jump() { ジャンプ }  
pass() { パス }  
punch() { パンチング }  
catch() { キャッチ }
```

以下のような課題が存在する：

- 歩く、走るなど処理内容が重複している
- 同じ「走る」という処理内容だが、メソッド名が異なってる
- FieldPlayerは、仲間にパスをすることができない…など

# 開発への取り組み方 2

事例：

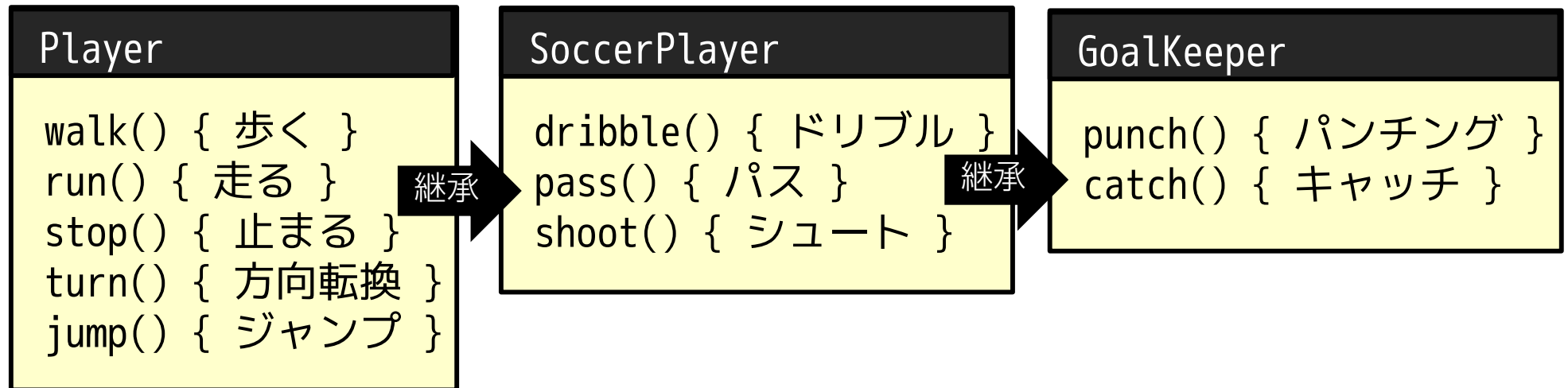
サッカーゲームの開発に関わることになった



取り組み方 2：

- Playerクラスを作成し、歩く、走る、止まる、方向転換、ジャンプのメソッドを実装した
- Playerクラスを継承したSoccerPlayerクラスを作成し、ドリブル、パス、シュートを追加実装した
- SoccerPlayerクラスを継承したGoalKeeperクラスを作成し、パンチングとキャッチを追加実装した

# 開発への取り組み方 2



取り組み方 1 との違い：

- 作成するクラスの数が多いが、同じ内容のメソッドが重複していない
- 直接的には利用しないクラス(Player)が存在する

# 取り組み方の比較

---

- 取り組み方1では、"無駄な" クラスを作成していないが、かえって作業に無駄が生じる可能性がある
  - 機能に重複や漏れが生じている
    - ⇒ 異なる人が同時にクラス作成に取り組む場合、同じ機能であっても、メソッド名や実装内容が異なる可能性がある
    - ⇒ GoalKeeperもドリブルやヘディング等を行うことがあるが、実装がされていない
- 取り組み方2は、長期的な視点に立っている
  - 将来的に総合スポーツゲームが作られることを想定し、あらゆるスポーツの基本動作を実装したPlayerクラスの作成からスタートしている
    - ⇒ 機能実装の重複や漏れを防ぐことができる
    - ⇒ 共通部が実装された親クラスを利用することで、チームでの開発がしやすいくなる

# 継承元クラスの必要性と課題

---

- 継承元となるクラスがあることで、機能実装における重複や漏れを防ぐことができ、チーム開発も行いやすくなる
- 一方で、継承元となる(継承を前提とする)クラスの作成は、必ずしも容易ではない
  - メソッドの具体的な内容を決定できない
  - 複数のクラスに分けて継承元を形成したいが、Javaでは多重継承を行うことはできない

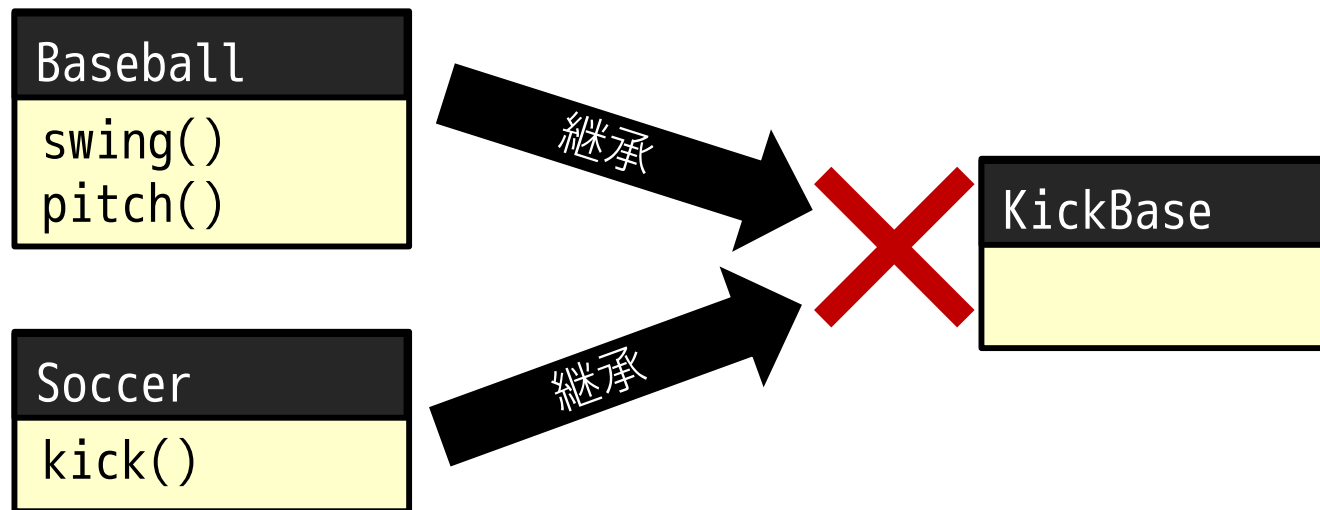
- 継承はis-aの関係で行われることが多く、継承が進むにつれ、より具体的なクラスになる  
⇒ 継承元を辿っていくと、より抽象的／汎用的になるためメソッドの具体的な内容が決定しづらい



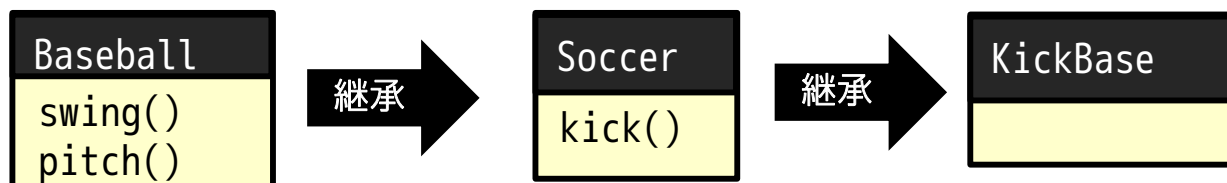
# 課題：多重継承ができない

- Javaでは、複数のクラスから同時に継承を行うことはできない

例：BaseballとSoccerを同時に継承するクラスの作成は不可



※ 複数のクラスを順番に継承するクラスの作成は可能



# 課題の解決策

- 内容の詳細が決定できないメソッドは、中身を空(から)にしておく方法が考えられる
  - 継承後にオーバーライドして詳細を記述させる

```
class Sports
public void score() {
}
```

中身を記述しない


継承

```
class Golf extends Sports
@Override
public void score() {
    // 全ホールの打数を合計
    .....
    // ハンディキャップの計算
    .....
}
```

継承後に具体的な内容を記述

# 課題の解決策

---

- 前頁の解決策では以下の問題が起こりうる
    - 意図に反して、継承を前提としているクラスを直接利用されてしまう
    - 継承先で、内容未確定のメソッドがオーバーライドされないまま利用されてしまう
- 
- Javaではこのような課題の解決手段として**抽象クラス**や**インターフェース**が用意されている
    - 継承とメソッドのオーバーライドが必須となる
    - インターフェースはクラスではないので多重継承を行うことができる  
(厳密には多重実装)

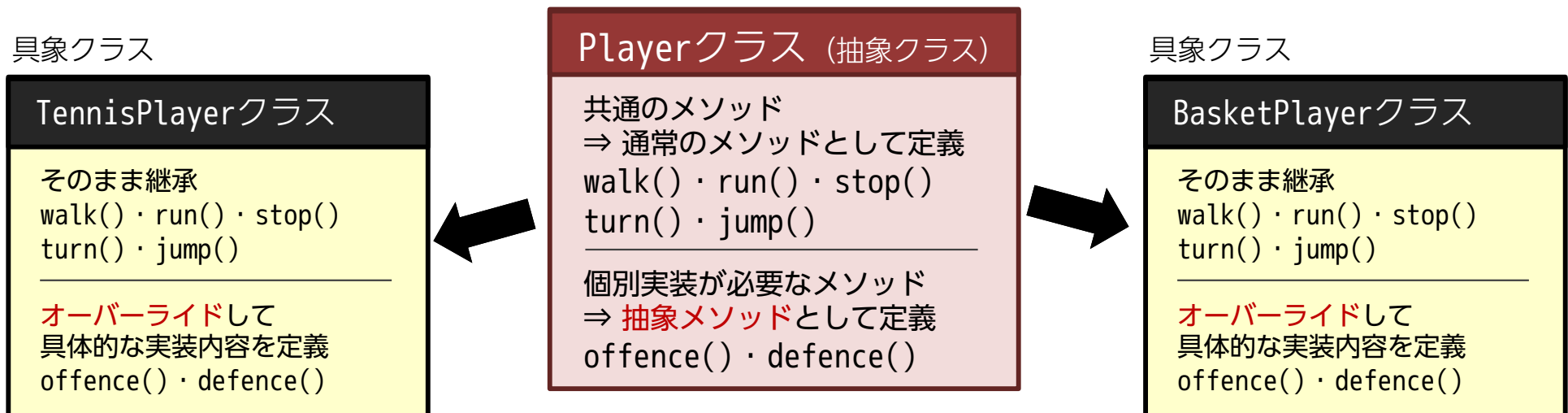
# 抽象クラス

---

- 具体的な処理内容／実装が記述されない**抽象メソッド**を持つクラスを**抽象クラス**という
    - 抽象メソッドはオーバーライドされることを前提としている
    - 抽象クラスは通常のメソッドも持つことができる
  - 抽象クラスからは直接インスタンスを生成することができない
    - 抽象クラスを継承したサブクラスを作り、そのサブクラスからインスタンスを生成する必要がある
- ⇒ 意図しない直接利用を防止することができる

# 抽象クラス

- 共通の処理を持つ複数のクラスを作りたい場面において、個別実装が必要なメソッドを明示的に指定できる
  - 抽象クラスを継承したクラス(具象クラス)を作成する必要がある
- 抽象クラスで定義されている抽象メソッドは必ずオーバーライドする必要がある
  - 抽象クラスを継承した別の抽象クラスを定義することも可能。その場合は抽象メソッドをオーバーライドする必要はない



# 抽象クラスの作成

- 抽象クラスの定義では、classキーワードの前に**abstract**を付ける
- 抽象メソッドは、戻り値の型の前に**abstract**を付ける
  - 具体的な処理の定義はせず、{ }の記述も不要

```
public abstract class Player {  
    protected int stamina = 100;  
    public void run() {  
        System.out.println("走ります");  
        stamina -= 10;  
    }  
    public abstract void offence();  
}
```

通常のフィールドやメソッド

抽象メソッド

# 抽象クラスの作成

- Eclipseで抽象クラスを作成する場合、abstractに✓を入れる

パッケージ(K):	practice22_1
<input type="checkbox"/> エンクロージング型(Y):	
名前(M):	Player
修飾子:	<input checked="" type="radio"/> public(P) <input type="radio"/> パッケージ <input checked="" type="checkbox"/> abstract(T) <input type="checkbox"/> final(L)
スーパークラス(S):	java.lang.Object
インターフェース(I):	

# 抽象クラスの継承

- 抽象クラスを継承したクラスでは、抽象メソッドをオーバーライドして実装する
  - 抽象メソッドをオーバーライドせずに、抽象クラスを継承した抽象クラスを作成することも可能

```
public class BasketPlayer extends Player {  
  
    @Override  
    public void offence() {  
        System.out.println("ボールをゴールに向けて投げます");  
        stamina -= 10;  
    }  
  
}
```

# 練習

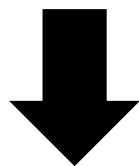
---

- 練習22-1
- 練習22-2

# より抽象的な抽象クラス

---

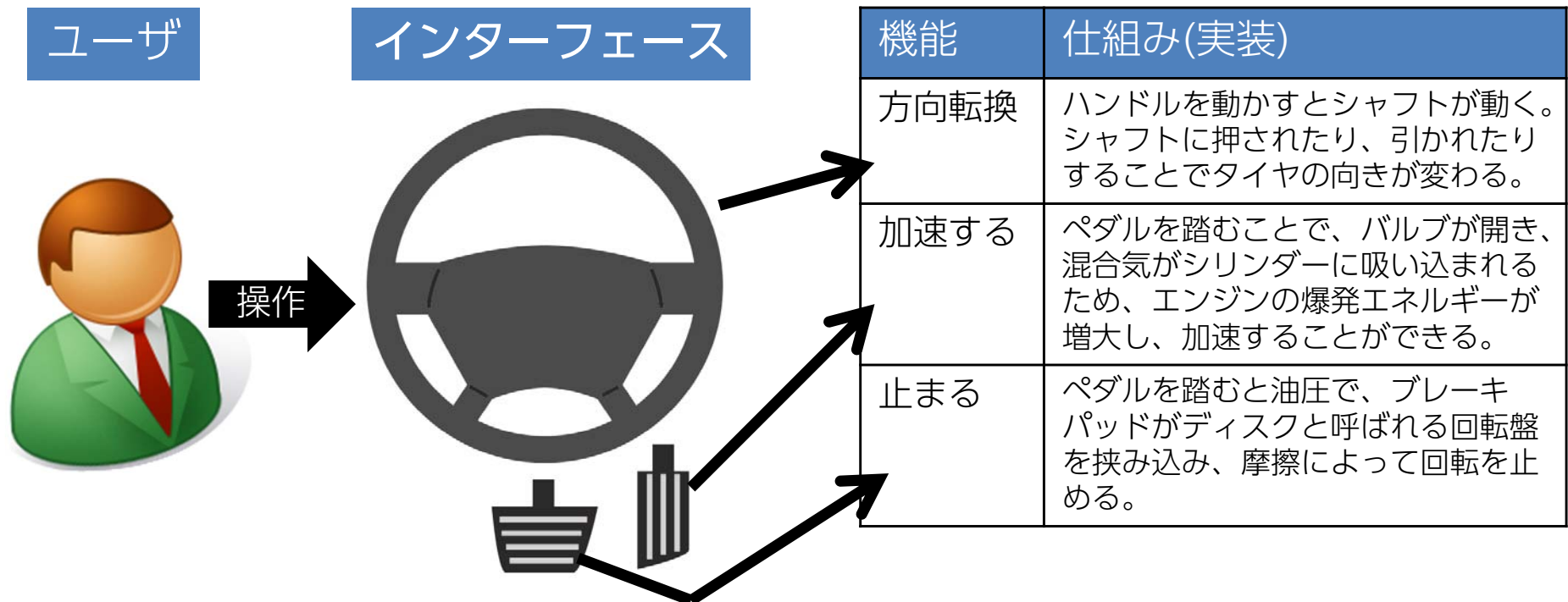
- 継承元を辿っていくと、より抽象的／汎用的なものになっていくため、抽象クラスが増える  
⇒ さらに辿っていくと、定義できる抽象メソッドやフィールドが減っていく



- 抽象メソッドのみのクラスは**インターフェース**として扱うことができる
  - 定数のフィールド(public static final)があってもよい
  - インターフェースとすることで、多重実装を行うことができる

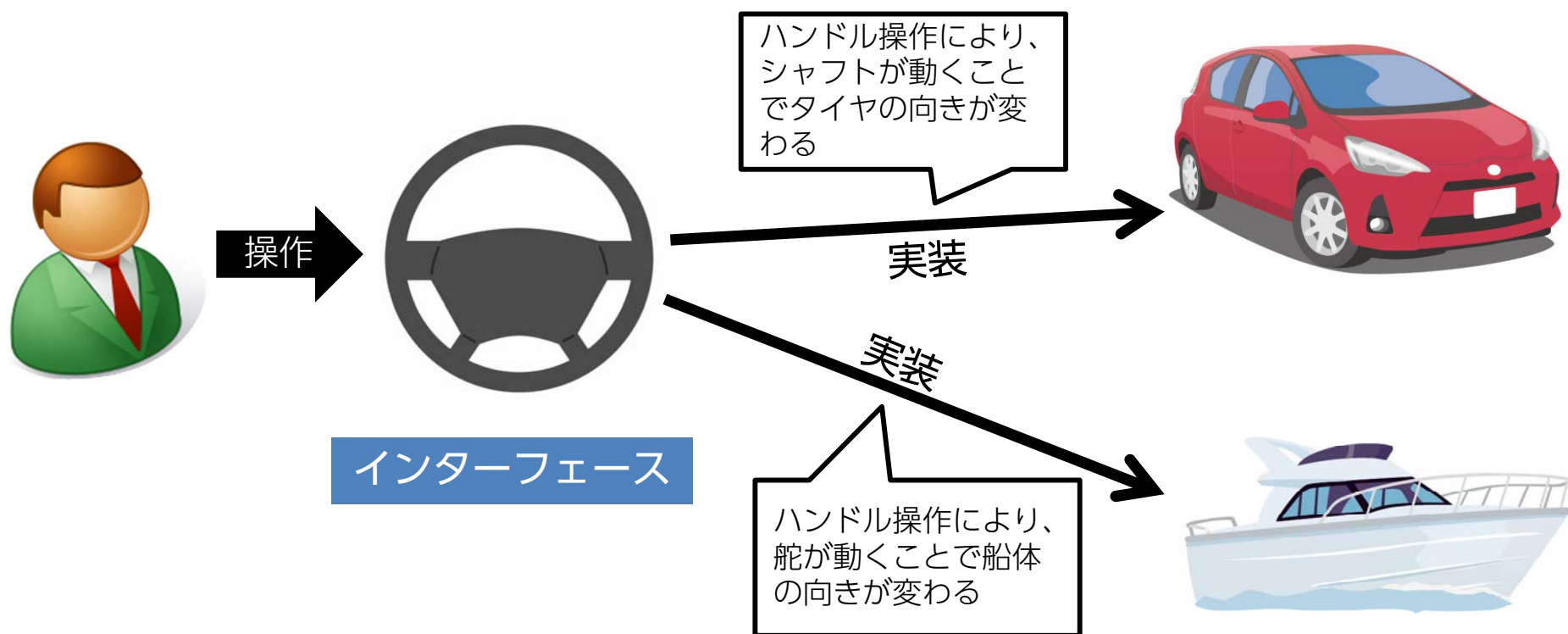
# インターフェースとは

- 接点や橋渡し役といった意味の言葉
  - ユーザと機能の接点／橋渡し役を担う
    - ⇒ ユーザはインターフェースを操作することで、機能にアクセスする
  - ユーザはインターフェースの操作方法だけ知っていればよい
    - ⇒ 機能の裏側にある「仕組み(実装)」を知る必要はない



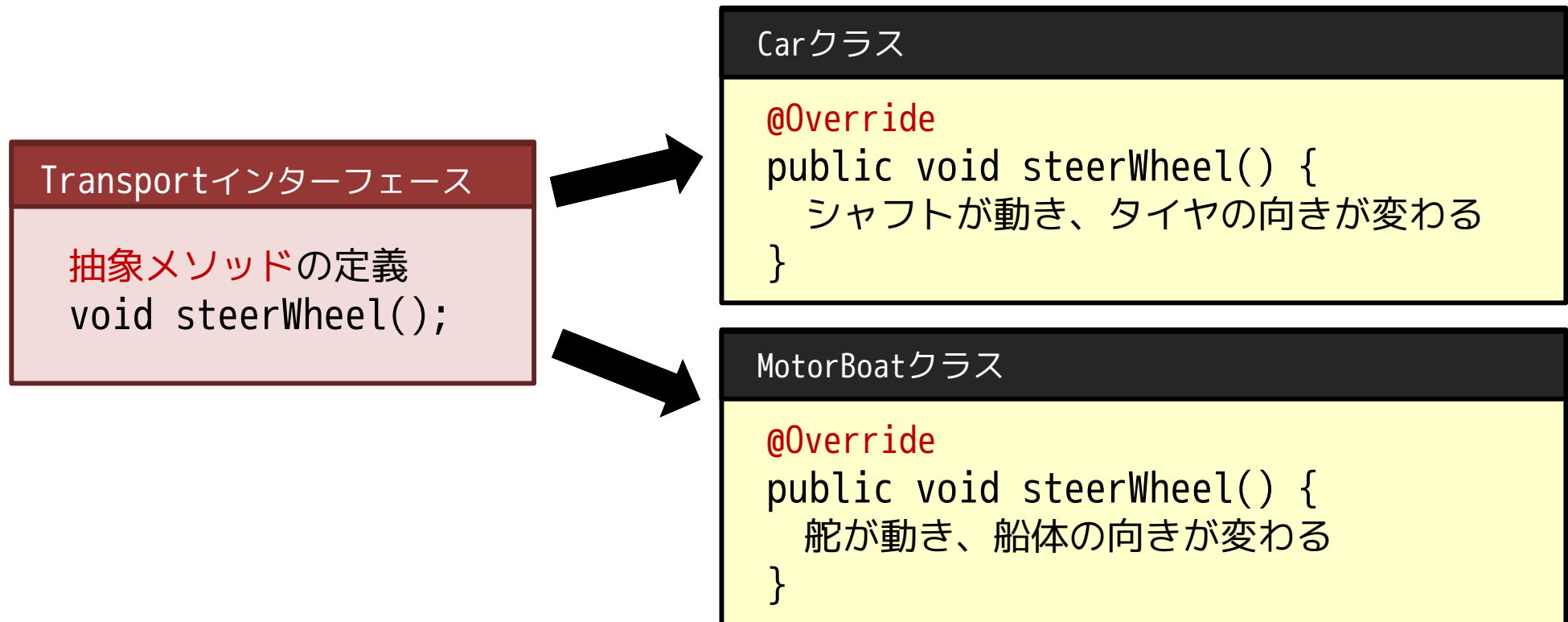
# インターフェースとは

- 共通のインターフェースをもつ(実装する)ものは、同じように操作することができる



# Javaのインターフェース

- クラスが実装すべきメソッドの名前、引数、戻り値の型のみを定義する仕組み
  - 具体的な処理内容は、インターフェースを実装したクラスに記述する



# Javaのインターフェース

- 同じインターフェースを持つ(=インターフェースを実装する)クラスのオブジェクトは同じように(統一されたメソッドを使って)操作することができる

```
public static void main(String[] args) {  
    Transport car = new Car();  
    Transport boat = new MotorBoat();  
  
    car.steerWheel();  
    boat.steerWheel();  
}
```

インターフェースの型に  
代入できる

異なるクラスのオブジェクトだが、  
同じ名前のメソッドで操作できる

# インターフェースの構文

- インターフェースの定義

```
アクセス修飾子 interface インターフェース名 {  
    戻り値の型 メソッド名(引数); // 自動的にpublicメソッドになる  
}
```

- インターフェースの実装

```
アクセス修飾子 class クラス名 implements インターフェース名 {  
    @Override  
    public 戻り値の型 メソッド名(引数) {  
        // インターフェースで定義されているメソッドの実装  
    }  
}
```

# インターフェースの例

- Transportインターフェース

```
public interface Transport {  
    void steerWheel();  
}
```

- Carクラス：Transportインターフェースを実装したクラス

```
public class Car implements Transport {  
    @Override  
    public void steerWheel() {  
        System.out.println("シャフトを動かし、タイヤの向きを変更");  
    }  
}
```

# インターフェース型変数

- インターフェースを実装したクラスのインスタンスは、そのインターフェースの型の変数に代入できる

Transportインターフェースを実装したCarクラスとMotorBoatクラスがある時、両方ともTransport型の変数に代入可能

```
Transport car = new Car();  
Transport boat = new MotorBoat();  
Transport[] transports = {car, boat};
```

本来は異なる型のオブジェクトを同じ型の変数に代入できる

# インターフェースの特徴

---

- フィールドとメソッドを持つことができる
- フィールドとメソッドには修飾子は不要
  - フィールドは常に **public static final** (定数)となる
  - メソッドは常に **public** な抽象メソッドとなる
- メソッドに処理コードは記述できない
  - defaultキーワードを付けると処理を記述できる
- 直接インスタンス化できない
  - インターフェースはクラスではない
    - ⇒ インターフェースを**実装**したクラスをインスタンス化する
- クラスと同様、ファイル名とインターフェース名は一致させる(例：Deviceインターフェース ⇒ Device.java)

# 練習

---

- 練習22-3
- 練習22-4

# 複数のインターフェースの実装

- クラスは複数のインターフェースを同時に実装すること(多重実装)ができる
  - 複数のクラスを同時に継承すること(多重継承)はできない

例：InputDeviceとOutputDeviceを実装したTouchPanelクラス

```
public class TouchPanel implements InputDevice, OutputDevice {  
    ...  
}
```

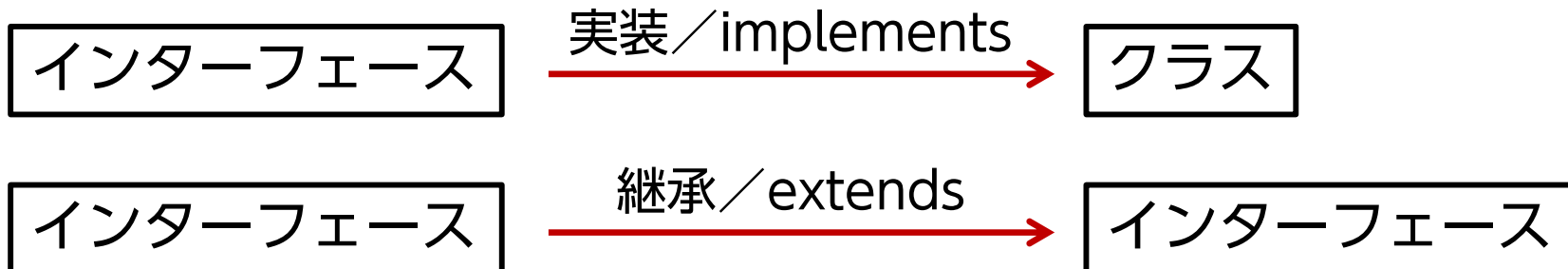
カンマ区切りで複数のインターフェースを列挙できる

# インターフェースの継承

- インターフェース同士であれば、クラスと同様に継承することができる
  - インターフェースは多重継承が可能

```
public interface IODevice extends InputDevice, OutputDevice {  
    ...  
}
```

## インターフェースの実装と継承



# マーカーインターフェース

- メンバーを持たないインターフェース
  - クラスの**種類**を示すためだけに使用する

例：PhisycalDevice (物理的デバイス) という種類を示すインターフェース

```
public interface PhisycalDevice {}
```



実装(または継承)して利用

```
public class Mouse implements PhisycalDevice {  
    ...  
}
```

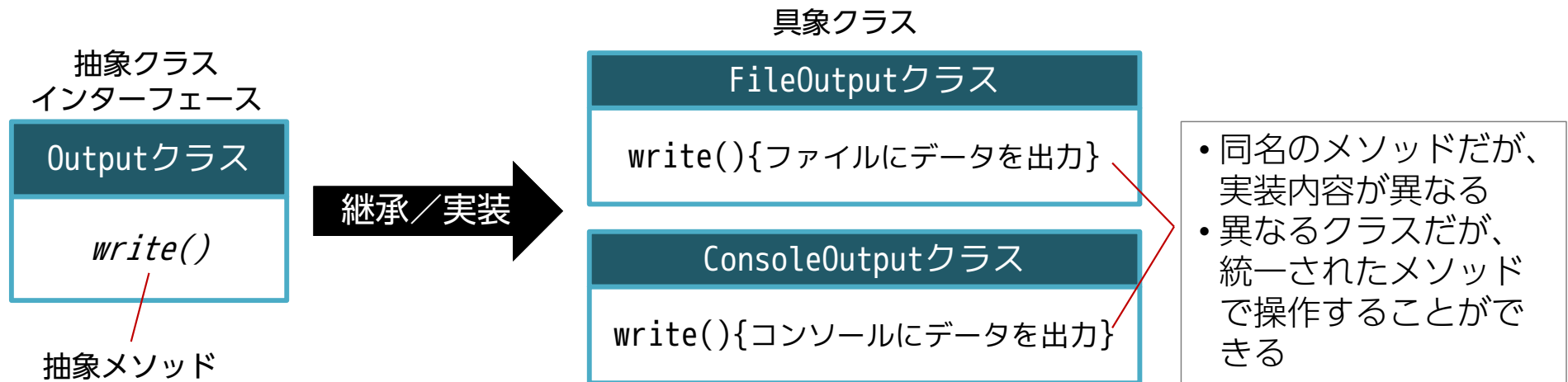
```
public class Keyboard implements PhisycalDevice {  
    ...  
}
```

# 抽象クラスとインターフェース

	抽象クラス	インターフェース
フィールド	通常のクラス同様、フィールドの定義を行える	定数のみ定義可能
メソッド	抽象メソッドと通常のメソッドの両方を定義可能	抽象メソッドとdefaultキーワード付きのメソッドを定義可能 ⇒ publicなメソッドになる
アクセス修飾子	通常のクラス同様に、アクセス修飾子を設定できる	アクセス修飾子の記述は不要 ⇒ 常にpublicとなる
多重継承	不可	複数のインターフェースを <b>実装</b> したクラスの作成が可能 複数のインターフェースを <b>継承</b> したインターフェースの作成が可能

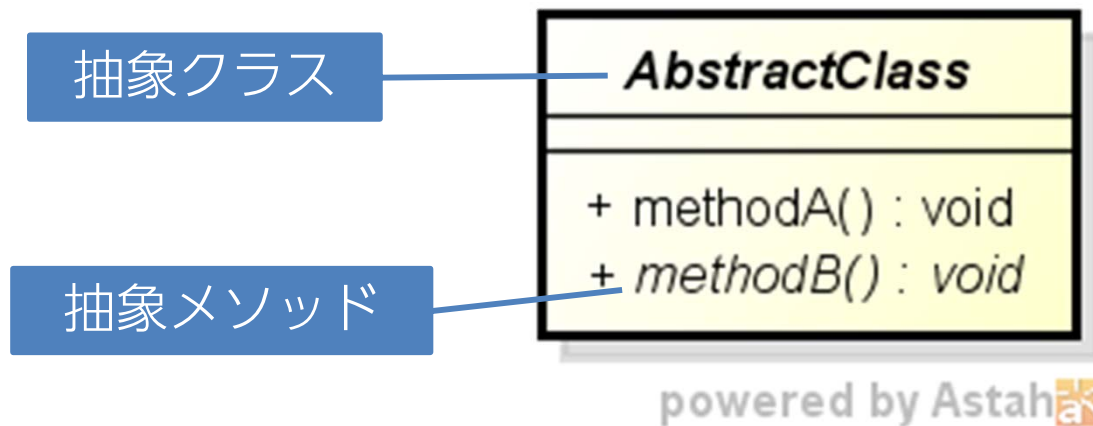
# 多態性

- 同じ名称のメソッドであっても、それを実装しているオブジェクトにより振る舞いが異なるという性質
- 抽象メソッドを利用することで、具体的な実装内容を記載することなく、メソッド名だけを決めることができる
  - 同じ抽象クラスやインターフェースを継承／実装したクラスであれば、統一されたメソッドで操作することができる



# UMLでの抽象クラスの表現

- 抽象クラス及び抽象メソッドは、斜体で示す

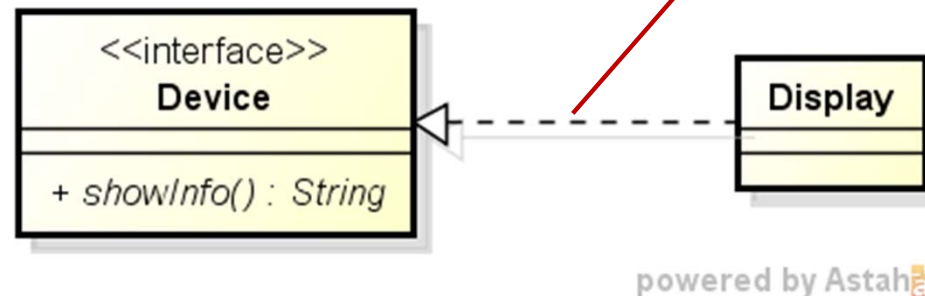


# クラス図でのインターフェースの表現

- <<interface>>というステレオタイプをつけたボックスで表す方法やロリポップを使った表記方法がある

クラス図で実装を表す記号  
クラス図の用語ではこの関連を**実現**と呼ぶ

ステレオタイプを使った表記



ロリポップを使った表記

