発展プログラミング

第1回：オブジェクト指向プログラミングの復習

宮田章裕 <miyata.akihiro@nihon-u.ac.jp>

学修環境

学修環境

- 第1回～第10回はJavaを用いる

- Java環境は学科配布のUbuntuを想定（OpenJDK 11）
  - 独自のJava環境（例：Mac，Windows上に構築したJava環境）の利用を禁止しない
  - ただし，独自環境の構築作業や，独自環境固有のトラブル対応はサポートしない

Java

- 本格的なオブジェクト指向プログラミング言語

- WORA
  - Write once, run anywhere
  - コンパイルされたJavaコードは，再コンパイルせずに他のプラットフォームで動作する

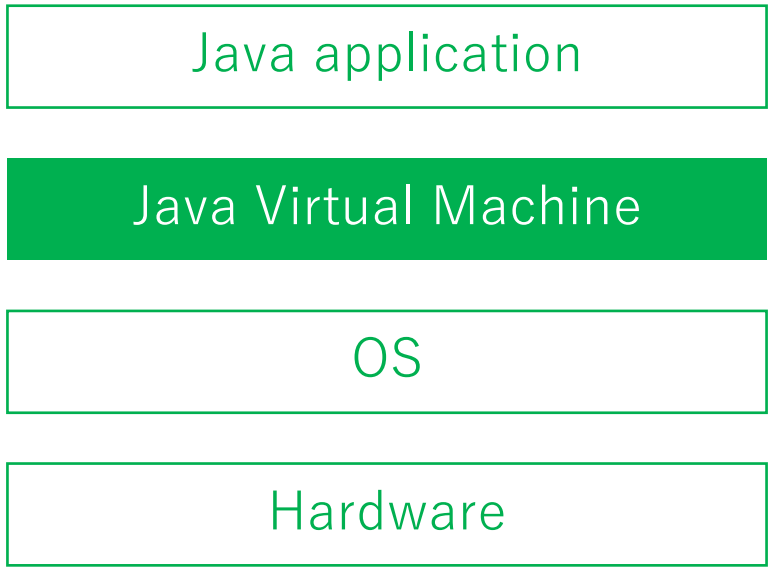
- 他言語との関係
  - C，C++と似た構文であるが，複雑な作業（例：ポインタ演算，メモリ管理）は排除されている
  - ProcessingはJavaで実装されており，Javaよりも簡易な記述で描画・アニメーションが行える

- 主な利用シーン
  - 企業におけるシステム開発
  - Androidアプリケーション開発

Java Virtual Machine (JVM)

❖Javaバイトコード（Javaソースファイルをコンパイルしたもの）を実行する環境

❖JVMがOS，ハードウェアの差を吸収するため，WORAが実現できる



Javaの動作確認

作業準備

- 本日の作業ディレクトリの作成・移動
  - mkdir -p SOMEWHERE/2021_ap/01
  - 以降，SOMEWHERE/2021_ap/01をWORK_DIRとする

- 「Javaの動作確認」の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - mkdir introduction
  - cd introduction

標準出力

- 「Hello, world!」と標準出力するプログラム
  - プログラムはmainメソッドから実行される
  - 標準出力
    - System.out.println(str)：strを標準出力する（出力後に改行あり）
    - System.out.print(str)： strを標準出力する（出力後に改行なし）

HelloWorld.java

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

実行結果

```
% javac HelloWorld.java
% java HelloWorld
Hello, world!
```

基本的な制御構文

❖Processingと同様

Syntax.java

```java
public class Syntax {
    public static void main(String[] args) {
        int age = 19;
        boolean withAdult = true;
        if(age >= 20 || withAdult) {
            System.out.println("You can enter.");
        } else {
            System.out.println("You cannot enter.");
        }

        int count = 3;
        while(count > 0) {
            System.out.println(count);
            count--;
        }

        int[] data = {1, 2, 3, 4, 5};
        int sum = 0;
        for(int i = 0; i < data.length; i++) {
            sum += data[i];
        }
        System.out.println(sum);
    }
}
```

実行結果

```
% javac Syntax.java
% java Syntax
You can enter.
3
2
1
15
```

授業内課題

- Bbの指定ページに時間内に提出すること
  - ㄨ切時刻を過ぎるとページが非表示になる
  - 授業内課題の提出物は授業参画度に反映する（成績の10%）
  - 授業内課題は正解・不正解は問わない（取り組んだ痕跡が認められればOK）

- 進捗状況をSlackで報告すること
  - Slackでの報告状況も授業参画度に反映する

IR01-1

❖空欄を埋めて，要件を満たすプログラムを作成せよ

(1)「Hello, world!」と10回標準出力するプログラム

```
public class Hello10 {
    public static void main(String[] args) {
        [空欄]
    }
}
```

(2) 1から10までの整数の和を標準出力するプログラム

```
public class Sum1to10 {
    public static void main(String[] args) {
        [空欄]
    }
}
```

(3) dataの各要素を先頭から順に，和が30以下である範囲で足し合わせ，その和を標準出力するプログラム

```
public class SumUntil30 {
    public static void main(String[] args) {
        int[] data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        [空欄]
    }
}
```

Javaの基本

作業準備

- 「Javaの基本」の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 01: Review > basic.zip をWORK_DIRにダウンロード
  - unzip basic.zip
  - cd basic

基本的なプログラム構成 (1/5)

Main.java

```java
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0);

        Printer printer = new Printer(msg, mark);
        printer.printMsg();
    }

}
```

Printer.java

```java


public class Printer {

    private String msg;
    private char mark;

    public Printer(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println(mark + msg);
    }

}
```

基本的なプログラム構成 (2/5)

Main.java

```java
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0);

        Printer printer = new Printer(msg, mark);
        printer.printMsg();
    }

}
```

クラス

mainメソッド

Printer.java

```java
public class Printer {

    private String msg;
    private char mark;

    public Printer(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println(mark + msg);
    }

}
```

クラス

メンバ変数

コンストラクタ

メソッド

- メンバ変数：クラス直下で定義された変数
- コンストラクタ：インスタンス作成時に呼び出されるメソッド

基本的なプログラム構成 (3/5)

Main.java

```
import java.util.*;   → Scannerを使うためのライブラリをインポート

public class Main {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in); → 標準入力を読み取るためのScannerインスタンスを作成

        System.out.print("Message: ");
        String msg = scan.nextLine(); → 標準入力全体をString型として読み込む

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0); → 標準入力の1文字目をchar型として読み込む

        Printer printer = new Printer(msg, mark); → Printerクラスのインスタンスを作成して
        printer.printMsg();                          Printerクラスのprint Msg()を呼び出す
    }

}
```

基本的なプログラム構成 (4/5)

Printer.java

```
public class Printer {

    private String msg;
    private char mark;

    public Printer(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println(mark + msg);
    }

}
```

→ メンバ変数を定義（privateについては後述）

→ コンストラクタで，引数で受け取った値をメンバ変数に代入。
msg，markはローカル変数（引数）を指す。
this.msg，this.markはメンバ変数を指す。

→ ここにはmsg，markという名前のローカル変数が無いので
msg，markはメンバ変数を指す。
this.msg，this.markとしても誤りではないが，
冗長なので普通はthis.は付けない。

基本的なプログラム構成 (5/5)

実行結果

```
% javac Main.java
% java Main
Message: Good morning
Mark: -
-Good morning
```

緑文字はキーボードからの標準入力

主な修飾子

- public
  - すべてのクラスからアクセス可能にする
- 修飾子なし（パッケージプライベート）
  - 同じパッケージ内と現在のクラスからアクセス可能にする
  - 本講義ではパッケージは扱わないので，積極的に用いなくてよい
- private
  - 同じクラスからのみアクセス可能にする
- static
  - クラスメソッド，クラス変数に指定する
- final
  - 再代入・オーバーライド（後述）を禁止する

IR01-2

- basic/Printer.javaに，下記の出力を行うための，printStrongerMsg()とprintStrongestMsg()を追加せよ。その後，Main.javaを適切に修正し，下記出力を得るプログラムを作成せよ。

目標の出力

```
% javac Main.java
% java Main
Message: Good morning
Mark: -

-Good morning                ← printMsg()による出力

---Good morning---           ← printStrongerMsg()による出力

------------
Good morning                 ← printStrongestMsg()による出力
------------
```

継承と抽象クラス

作業準備

- 「継承と抽象クラス」の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 01: Review > inheritance.zip をWORK_DIRにダウンロード
  - unzip inheritance.zip
  - cd inheritance

継承・抽象クラスの必要性 (1/3)

Main.java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0);

        SideDecorator sdDecorator
            = new SideDecorator(msg, mark);
        sdDecorator.printMsg();

        UpBottomDecorator ubDecorator
            = new UpBottomDecorator(msg, mark);
        ubDecorator.printMsg();
    }
}
```

実行結果

```
Message: Good morning
Mark: -

SideDecorator: printMsg()
-Good morning-

UpBottomDecorator: printMsg()
------------
Good morning
------------
```

SideDecorator.java

```java
public class SideDecorator {
    private String msg;
    private char mark;

    public SideDecorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "SideDecorator: printMsg()");
        printMarks(1, false);
        System.out.print(msg);
        printMarks(1, true);
    }

    private void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }
}
```

UpBottomDecorator.java

```java
public class UpBottomDecorator {
    private String msg;
    private char mark;

    public UpBottomDecorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "UpBottomDecorator: printMsg()");
        printMarks(msg.length(), true);
        System.out.println(msg);
        printMarks(msg.length(), true);
    }

    private void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }
}
```

継承・抽象クラスの必要性 (2/3)

Main.java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0);

        SideDecorator sdDecorator
            = new SideDecorator(msg, mark);
        sdDecorator.printMsg();

        UpBottomDecorator ubDecorator
            = new UpBottomDecorator(msg, mark);
        ubDecorator.printMsg();
    }
}
```

SideDecorator.java

```java
public class SideDecorator {
    private String msg;
    private char mark;

    public SideDecorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "SideDecorator: printMsg()");
        printMarks(1, false);
        System.out.print(msg);
        printMarks(1, true);
    }

    private void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }
}
```

UpBottomDecorator.java

```java
public class UpBottomDecorator {
    private String msg;
    private char mark;

    public UpBottomDecorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "UpBottomDecorator: printMsg()");
        printMarks(msg.length(), true);
        System.out.println(msg);
        printMarks(msg.length(), true);
    }

    private void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }
}
```

同じ処理が複数クラスで
重複している

継承・抽象クラスの必要性 (3/3)

Main.java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0);

        SideDecorator sdDecorator
            = new SideDecorator(msg, mark);
        sdDecorator.printMsg();

        UpBottomDecorator ubDecorator
            = new UpBottomDecorator(msg, mark);
        ubDecorator.printMsg();
    }
}
```

SideDecoratorクラスとUpBottomDecoratorクラスがprintMsg()を実装していることが保証されていない

SideDecorator.java

```java
public class SideDecorator {
    private String msg;
    private char mark;

    public SideDecorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "SideDecorator: printMsg()");
        printMarks(1, false);
        System.out.print(msg);
        printMarks(1, true);
    }

    private void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }
}
```

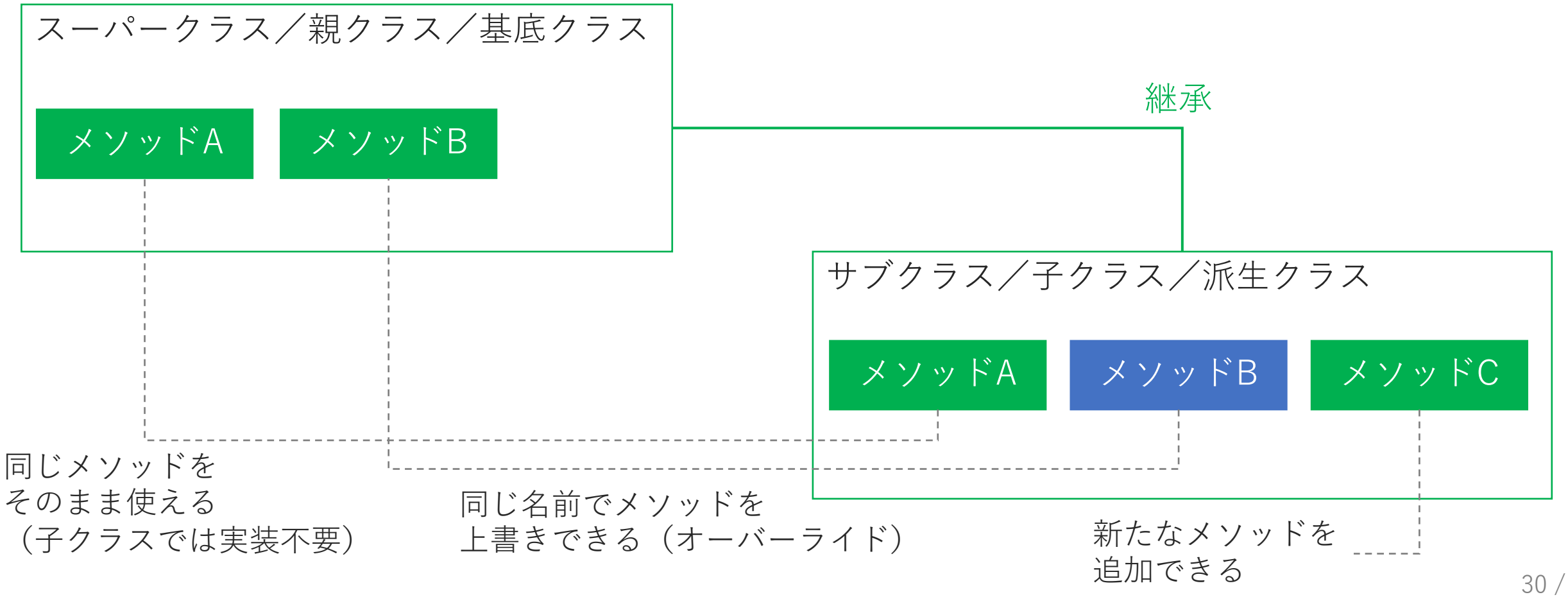
UpBottomDecorator.java

```java
public class UpBottomDecorator {
    private String msg;
    private char mark;

    public UpBottomDecorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "UpBottomDecorator: printMsg()");
        printMarks(msg.length(), true);
        System.out.println(msg);
        printMarks(msg.length(), true);
    }

    private void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }
}
```

現時点ではクラス数が少ないし，各クラスが短いので，printMsg()が実装されているか否かの確認は容易。しかし，クラス数が増え，各クラスの記述量が増えた場合はこの限りではない。

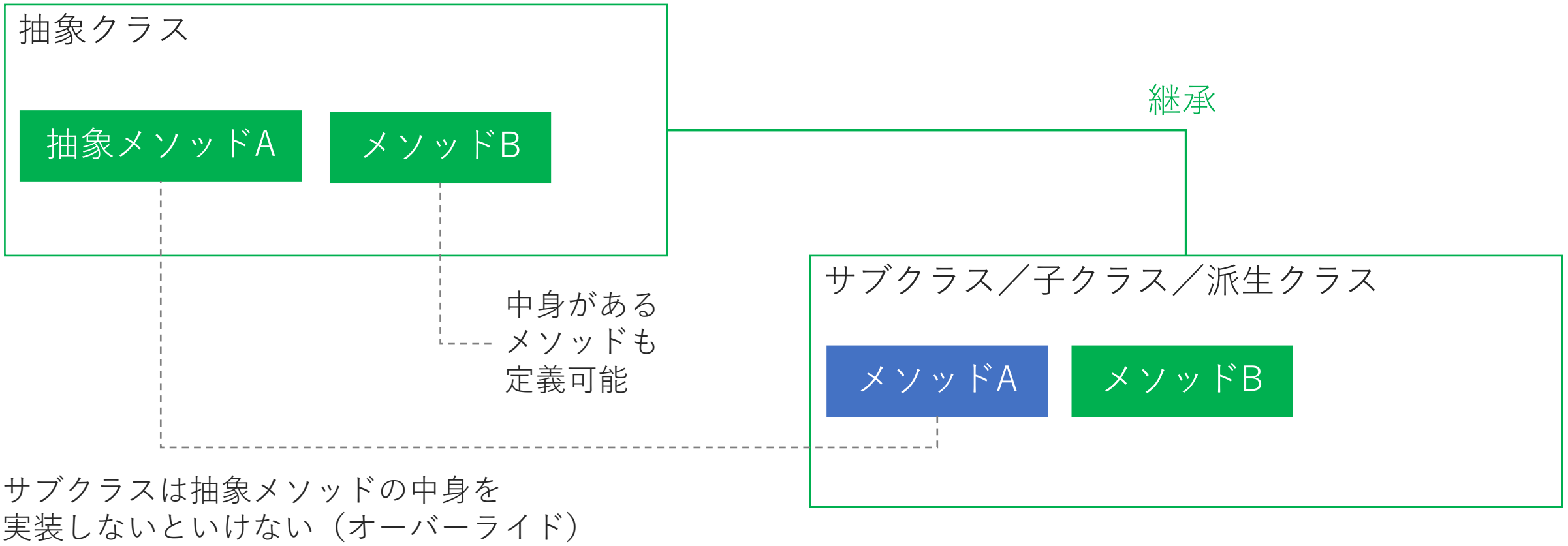
継承

- 元になるクラスのメソッド，メンバ変数を引き継ぎながら，新たな機能を加えたり，元の機能を上書きしたりする仕組み



抽象クラス

- 1つ以上の抽象メソッド（中身が空のメソッド）を含み，サブクラスに抽象メソッドの実装を義務付けられるクラス



継承・抽象クラスの利用例 (1/3)

Main.java

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0);

        Decorator sdDecorator
            = new SideDecorator(msg, mark);
        sdDecorator.printMsg();

        Decorator ubDecorator
            = new UpBottomDecorator(msg, mark);
        ubDecorator.printMsg();
    }
}
```

Decorator.java（抽象クラス）

```
public abstract class Decorator {
    public String msg;
    public char mark;

    public Decorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }

    public abstract void printMsg();
}
```

SideDecorator.java

```
public class SideDecorator extends Decorator {
    public SideDecorator(String msg, char mark) {
        super(msg, mark);
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "SideDecorator: printMsg()");
        printMarks(1, false);
        System.out.print(msg);
        printMarks(1, true);
    }
}
```

UpBottomDecorator.java

```
public class UpBottomDecorator extends Decorator {
    public UpBottomDecorator(String msg, char mark) {
        super(msg, mark);
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "UpBottomDecorator: printMsg()");
        printMarks(msg.length(), true);
        System.out.println(msg);
        printMarks(msg.length(), true);
    }
}
```

継承・抽象クラスの利用例 (2/3)

Main.java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        System.out.print("Mark: ");
        char mark = scan.next().charAt(0);

        Decorator sdDecorator
            = new SideDecorator(msg, mark);
        sdDecorator.printMsg();

        Decorator ubDecorator
            = new UpBottomDecorator(msg, mark);
        ubDecorator.printMsg();
    }
}
```

Decoratorは抽象クラスであるため
インスタンスの作成ができないので
Decoratorを継承したSideDecoratorクラス，UpBottomDecoratorクラスのインスタンスを作成。
Decoratorを継承しているので，各クラスのインスタンスからは必ずprintMsg()を呼び出せる。

Decorator.java

抽象クラスにはabstractを付ける

```java
public abstract class Decorator {

    public String msg;
    public char mark;

    public Decorator(String msg, char mark) {
        this.msg = msg;
        this.mark = mark;
    }

    public void printMarks(
        int count, boolean linefeed) {
        for(int i = 0; i < count; i++) {
            System.out.print(mark);
        }
        if(linefeed) {
            System.out.println();
        }
    }

    public abstract void printMsg();
}
```

サブクラスから
利用するので
publicにする

サブクラスに
実装が義務付けられた
抽象メソッド

継承・抽象クラスの利用例 (3/3)

SideDecorator.java

```
public class SideDecorator extends Decorator {
    public SideDecorator(String msg, char mark) {

        super(msg, mark);

    }


    public void printMsg() {
        System.out.println();
        System.out.println(
            "SideDecorator: printMsg()");
        printMarks(1, false);
        System.out.print(msg);
        printMarks(1, true);
    }
}
```

UpBottomDecorator.java

```
public class UpBottomDecorator extends Decorator {
    public UpBottomDecorator(String msg, char mark) {

        super(msg, mark);

    }


    public void printMsg() {
        System.out.println();
        System.out.println(
            "UpBottomDecorator: printMsg()");
        printMarks(msg.length(), true);
        System.out.println(msg);
        printMarks(msg.length(), true);
    }
}
```

Decoratorクラスを継承

スーパークラスの
コンストラクタを
呼び出す

抽象クラスであるスーパークラスに
実装を義務付けられたprintMsg()を実装

スーパークラスの
メソッドやメンバ変数を利用可能

IR01-3

- 下記の出力を行うための，SurroundingDecoratorクラスを実装せよ。その後，Main.javaを適切に修正し，下記出力を得るプログラムを作成せよ。なお，String型変数strの文字列長はstr.length()で求められる。

目標の出力

```
% javac Main.java
% java Main
Message: Good morning
Mark: -

SideDecorator: printMsg()
-Good morning-

UpBottomDecorator: printMsg()
------------
Good morning
------------

SurroundingDecorator: printMsg()
--------------
-Good morning-
--------------
```

インタフェース

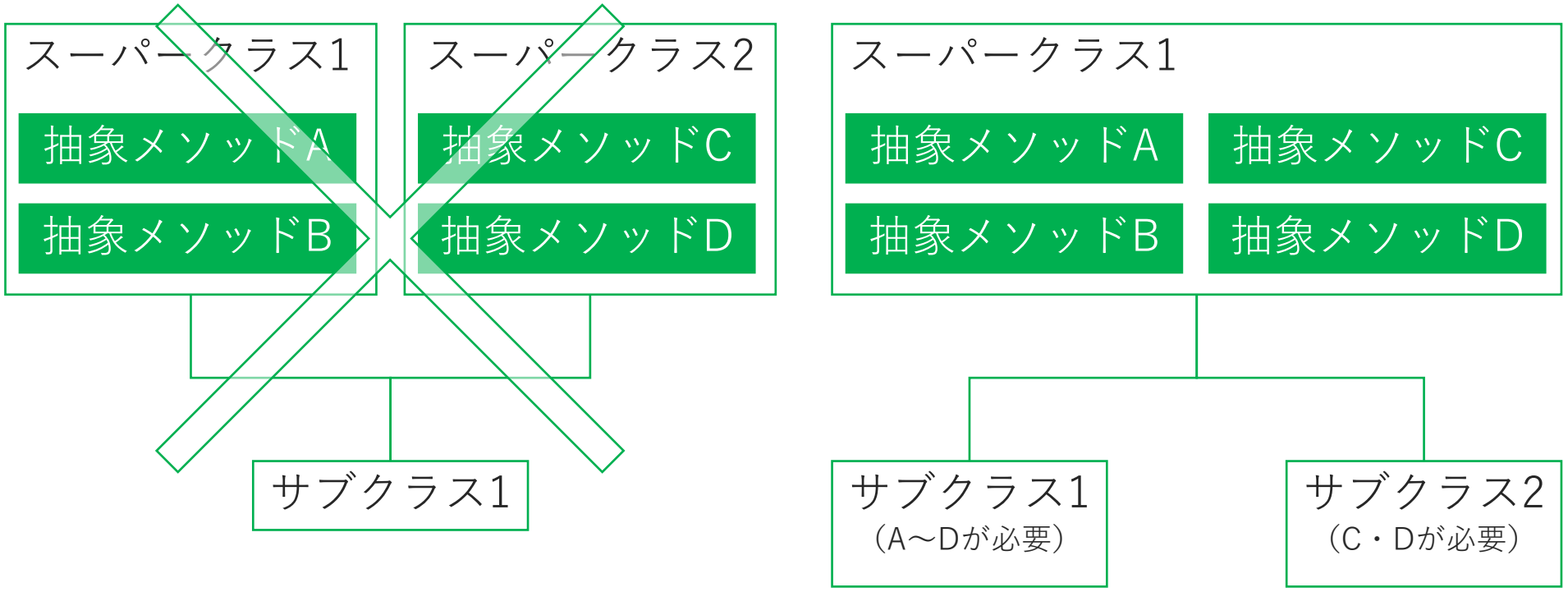
作業準備

- 「インタフェース」の作業ディレクトリの作成・移動
  - cd WORK_DIR
  - Bb > 01: Review > interface.zip をWORK_DIRにダウンロード
  - unzip interface.zip
  - cd interface

インタフェースの必要性

- 抽象クラスの導入により，サブクラスに指定メソッドの実装を義務付けることが可能になった

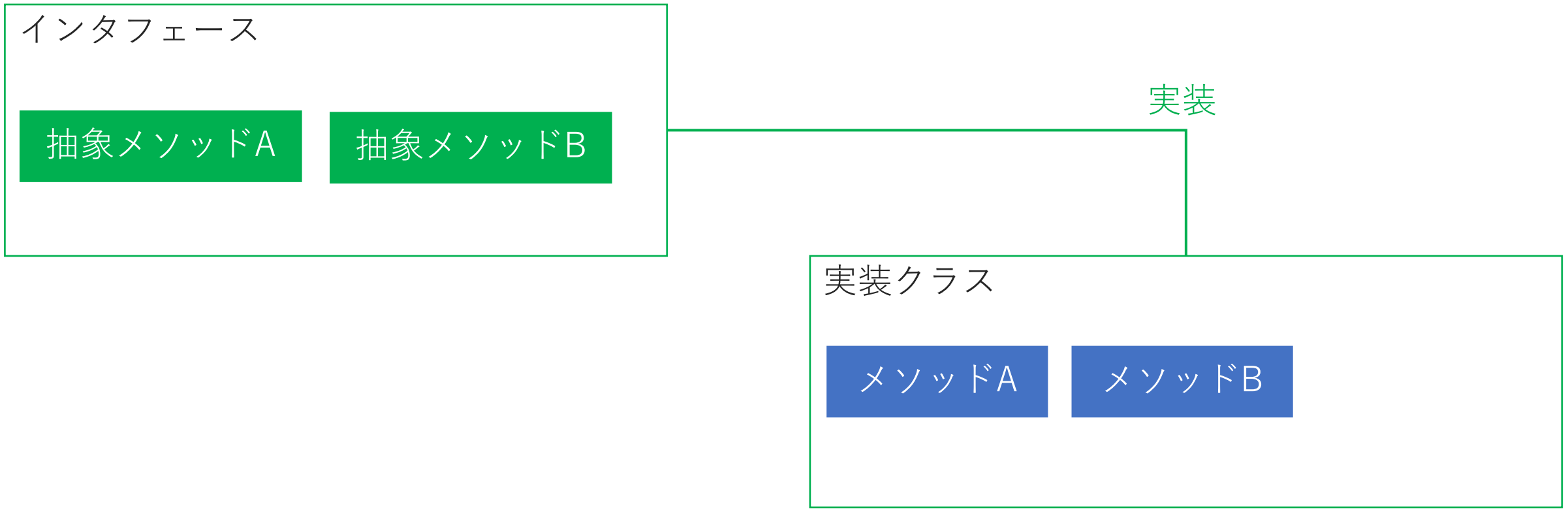
- しかし，Javaでは1つのスーパークラスしか継承できない（多重継承の禁止）



多重継承を避けるために
サブクラス1に必要なメソッドを全部
スーパークラス1に入れてしまうと
サブクラス2は余計なメソッドまで
オーバーライドしなくてはいけない

インタフェース

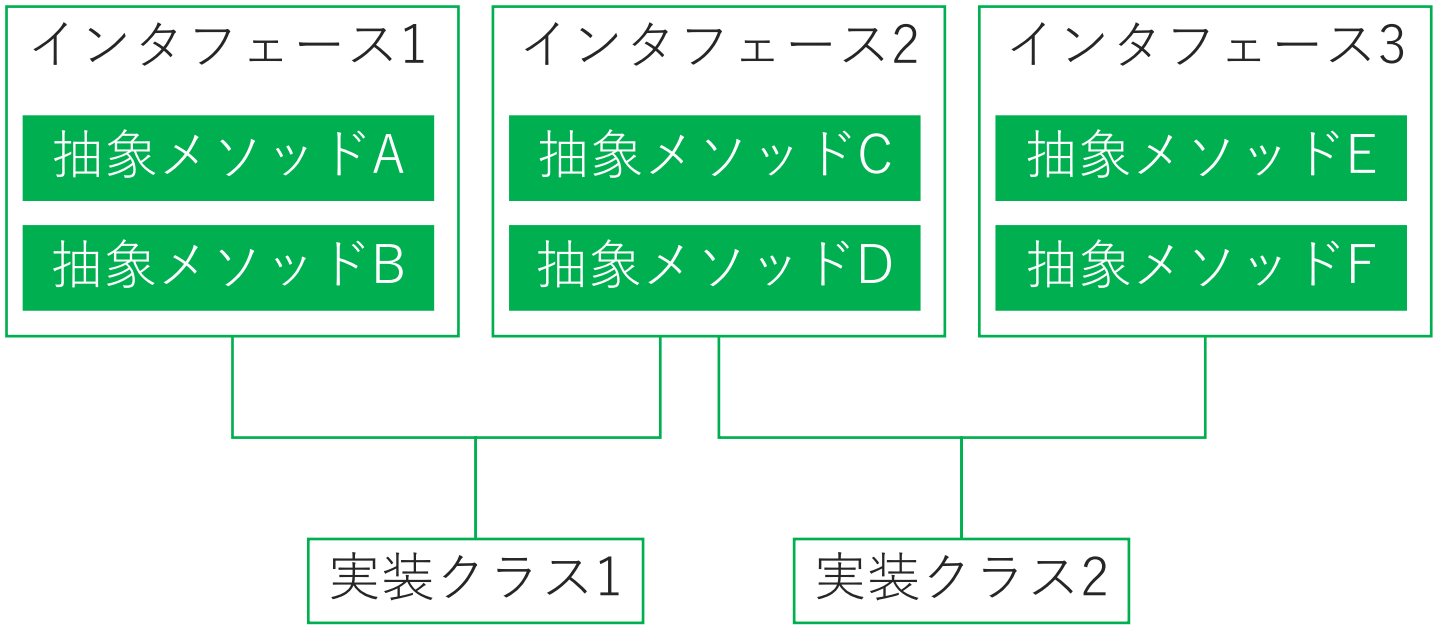
- メソッドが全て抽象メソッド（中身が空のメソッド）であり，実装クラスに抽象メソッドの実装を義務付けられるクラス



実装クラスは抽象メソッドの中身を
実装しないといけない（オーバーライド）

インタフェースは複数実装可能

- 1つのクラスが，複数のインタフェースを実装可能
- これにより，適切な単位でインタフェースを分割し，各実装クラスで必要なインタフェースを取捨選択して実装できる



インタフェースの利用例 (1/2)

Main.java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        Printable markPrinter
            = new MarkPrinter(msg);
        markPrinter.printMsg();
        markPrinter.printStrongMsg();

        Printable casePrinter
            = new CasePrinter(msg);
        casePrinter.printMsg();
        casePrinter.printStrongMsg();
    }
}
```

MarkPrinter.java

```java
public class MarkPrinter implements Printable {
    private final char MARK = '!';
    private String msg;

    public MarkPrinter(String msg) {
        this.msg = msg;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "MarkPrinter: printMsg()");
        System.out.println(msg);
    }

    public void printStrongMsg() {
        System.out.println();
        System.out.println(
            "MarkPrinter: printStrongMsg()");
        System.out.println(msg + MARK);
    }
}
```

CasePrinter.java

```java
public class CasePrinter implements Printable {
    private String msg;

    public CasePrinter(String msg) {
        this.msg = msg;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "CasePrinter: printMsg()");
        System.out.println(msg);
    }


    public void printStrongMsg() {
        System.out.println();
        System.out.println(
            "CasePrinter: printStrongMsg()");
        System.out.println(msg.toUpperCase());
    }
}
```

Printable.java（インタフェース）

```java
public interface Printable {
    void printMsg();
    void printStrongMsg();
}
```

インタフェースの利用例 (2/2)

Printableクラスを実装

Main.java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Message: ");
        String msg = scan.nextLine();

        Printable markPrinter
            = new MarkPrinter(msg);
        markPrinter.printMsg();
        markPrinter.printStrongMsg();

        Printable casePrinter
            = new CasePrinter(msg);
        casePrinter.printMsg();
        casePrinter.printStrongMsg();
    }
}
```

インタフェースを実装したクラスの
インスタンスを作成

MarkPrinter.java

```java
public class MarkPrinter implements Printable {

    private final char MARK = '!';
    private String msg;

    public MarkPrinter(String msg) {
        this.msg = msg;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "MarkPrinter: printMsg()");
        System.out.println(msg);
    }

    public void printStrongMsg() {
        System.out.println();
        System.out.println(
            "MarkPrinter: printStrongMsg()");
        System.out.println(msg + MARK);
    }
}
```

CasePrinter.java

```java
public class CasePrinter implements Printable {

    private String msg;

    public CasePrinter(String msg) {
        this.msg = msg;
    }

    public void printMsg() {
        System.out.println();
        System.out.println(
            "CasePrinter: printMsg()");
        System.out.println(msg);
    }


    public void printStrongMsg() {
        System.out.println();
        System.out.println(
            "CasePrinter: printStrongMsg()");
        System.out.println(msg.toUpperCase());
    }
}
```

Printable.java（インタフェース）

```java
public interface Printable {
    void printMsg();
    void printStrongMsg();
}
```

メソッドは抽象メソッドのみ
（中身のあるメソッドは実装できない）

IR01-4（任意）

- Printableインタフェースを実装したクラスを1つ追加せよ。その後，Main.javaを適切に修正し，新たなクラスで実装したprintMsg()，printStrongMsg()を呼び出すプログラムを作成せよ。

本日のまとめ

- 講義内容
  - 学修環境の確認
  - Javaの動作確認
  - Javaの基本
  - 継承と抽象クラス
  - インタフェース

- 授業内課題提出
  - 各授業内課題（IR）の解答を記載せよ。
  - 「講義内容のまとめ」の解答欄に
    上記「講義内容」の各項目について文章で説明を記載せよ。