

発展プログラミング

第2回：デザインパターンの導入

宮田章裕 <miyata.akihiro@nihon-u.ac.jp>

前回講義の復習

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

```
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

```
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
```

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

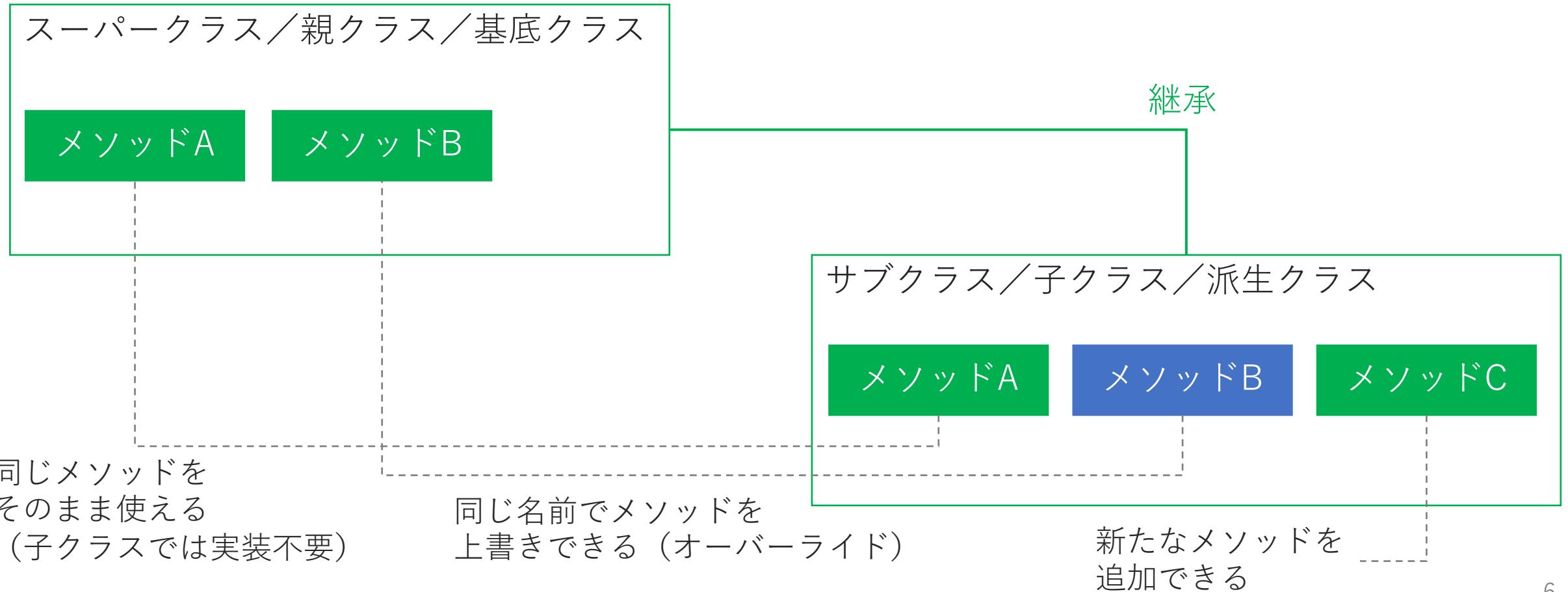
- ❖ クラスメソッド、クラス変数に指定する

❖ final

- ❖ 再代入・オーバーライド（後述）を禁止する

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



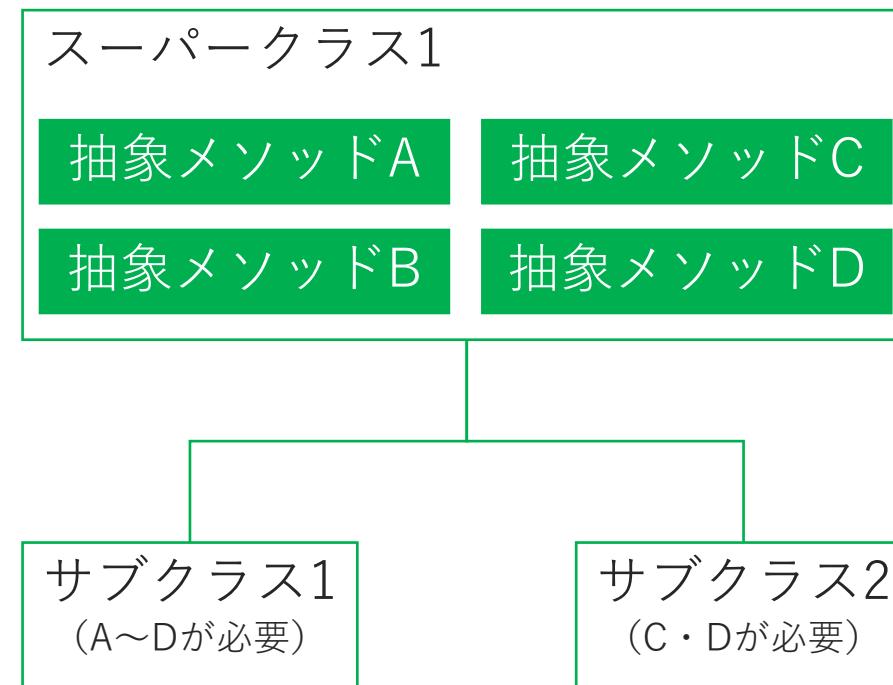
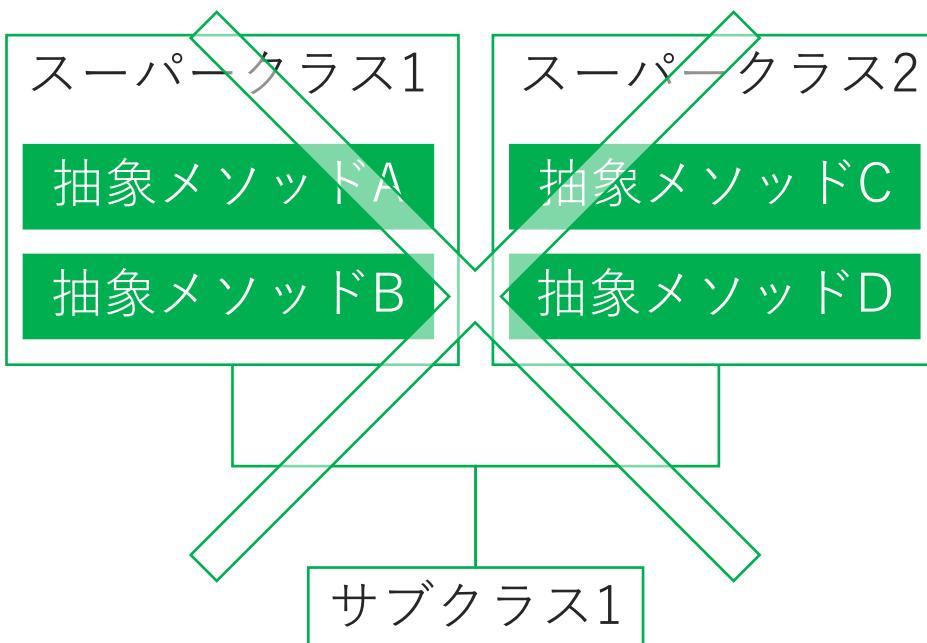
抽象クラス

- ❖ 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



インタフェースの必要性

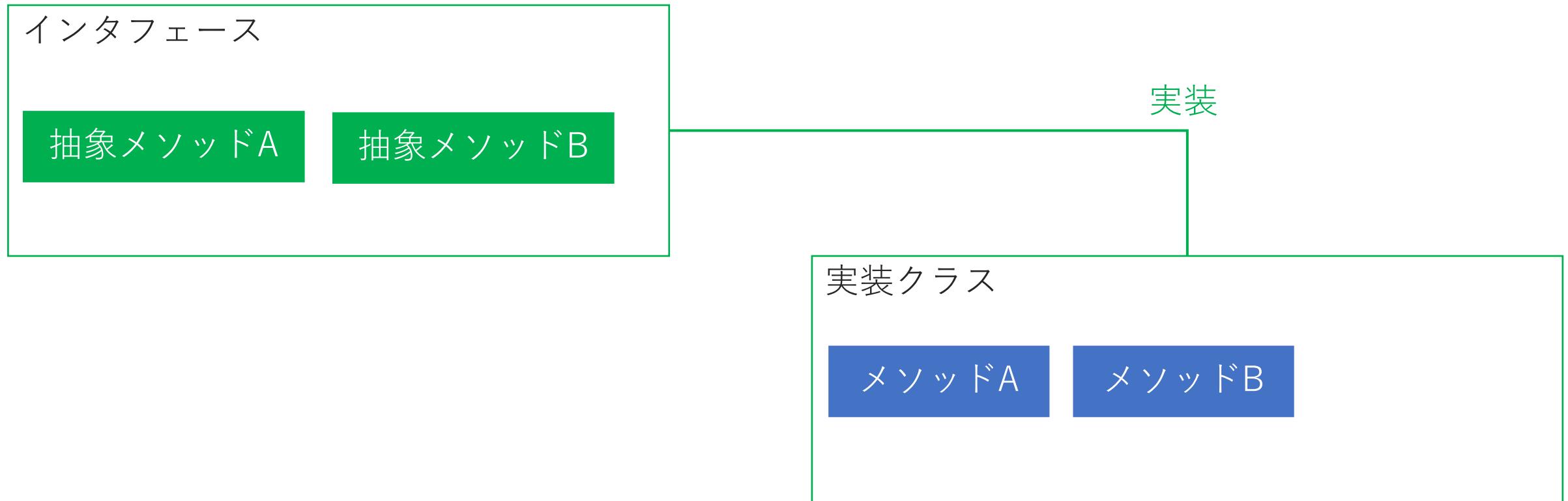
- ❖ 抽象クラスの導入により、サブクラスに指定メソッドの実装を義務付けることが可能になった
- ❖ しかし、Javaでは1つのスーパークラスしか継承できない（多重継承の禁止）



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

インターフェース

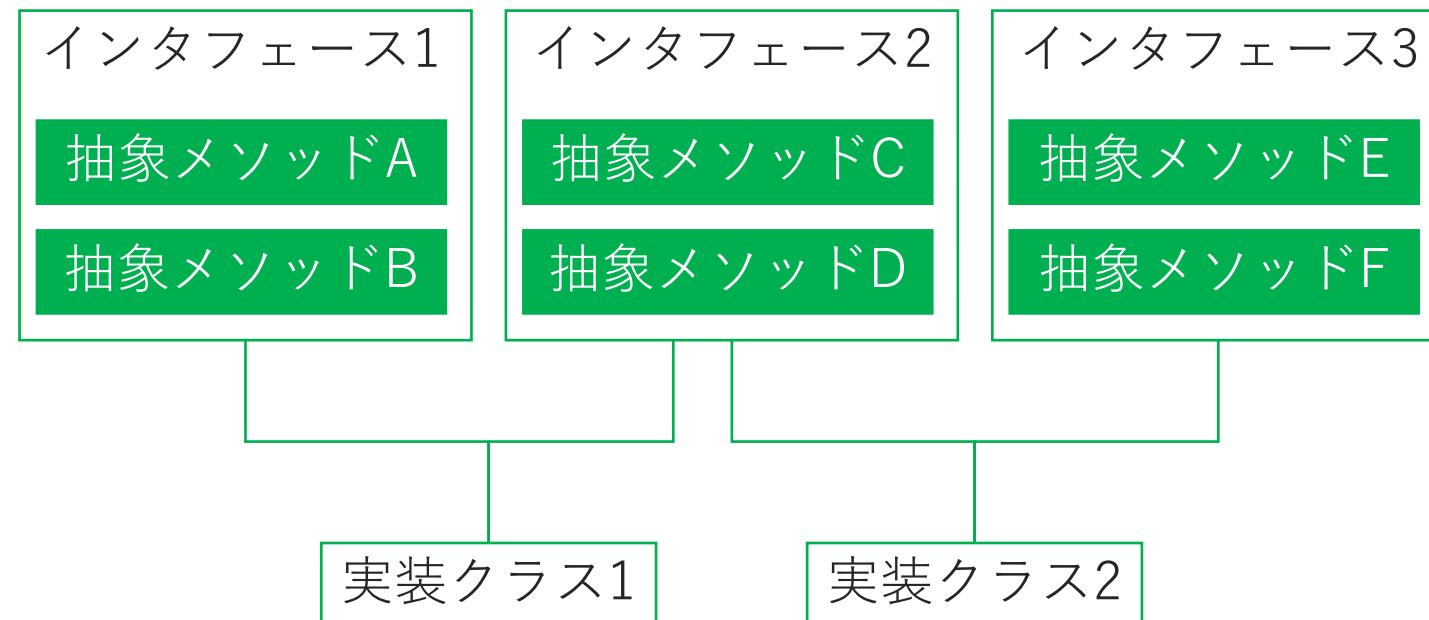
- ❖ メソッドが全て抽象メソッド（中身が空のメソッド）であり、実装クラスに抽象メソッドの実装を義務付けられるクラス



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）

インターフェースは複数実装可能

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



“デザインパターンとは

デザインパターン

- ❖ the Gang of Four (GoF: E. Gamma, R. Helm, R. Johnson, J. Vlissiders) が整理した、プログラミングにおいて頻出するパターン
- ❖ デザインパターンを学ぶ主目的：再利用しやすいコードを作成できるようになる
- ❖ あるいは、既存コードを再利用する（せざるをえない）ため、
という理由の方が学生のみなさんには共感しやすいかもしれない
 - ❖ 例1：あるコード群がライブラリとして提供されていて、自分にはそれらの変更権限がない
 - ❖ 例2：試験が済んだ既存コード群があり、そこに手を入れると試験がやり直しになってしまう

デザインパターンの種類

- ❖ GoFは23個のデザインパターンを定義
- ❖ 本講義では重要かつ使用頻度が高いものを厳選して説明する

| | |
|------------------|-------------------------|
| Iterator | Visitor |
| Adapter | Chain of Responsibility |
| Template Method | Facade |
| Factory Method | Mediator |
| Singleton | Observer |
| Prototype | Memento |
| Builder | State |
| Abstract Factory | Flyweight |
| Bridge | Proxy |
| Strategy | Command |
| Composite | Interpreter |
| Decorator | |

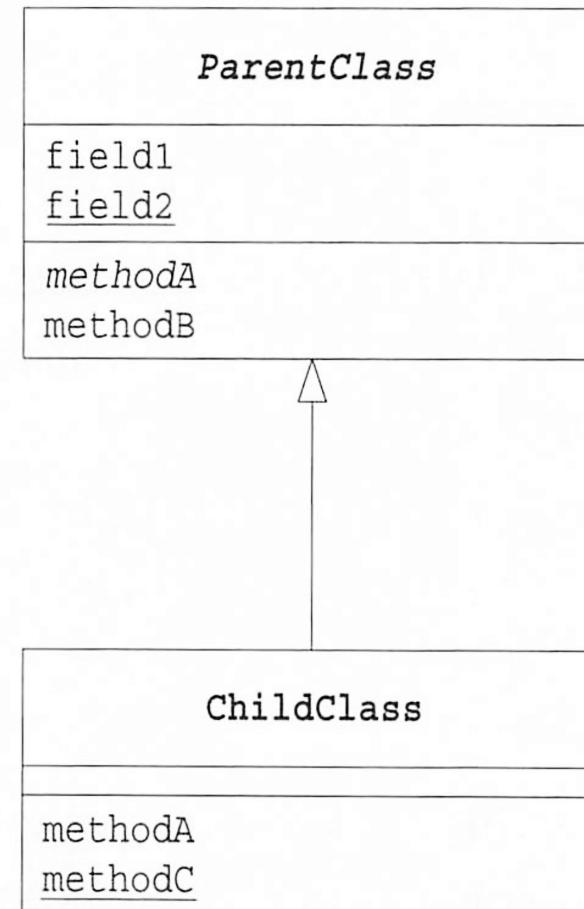
UML

- ❖ システムを視覚化したり、仕様や設計を文書化したりするための表現方法
- ❖ 本講義では、Javaのクラス間の関係（クラス図）を示すために用いる

UMLによるクラス図 (1/4)

❖ 繙承

```
abstract class ParentClass {  
    int field1;  
    static char field2;  
    abstract void methodA();  
    double methodB() {  
        // ...  
    }  
}  
  
class ChildClass extends ParentClass {  
    void methodA() {  
        // ...  
    }  
    static void methodC() {  
        // ...  
    }  
}
```

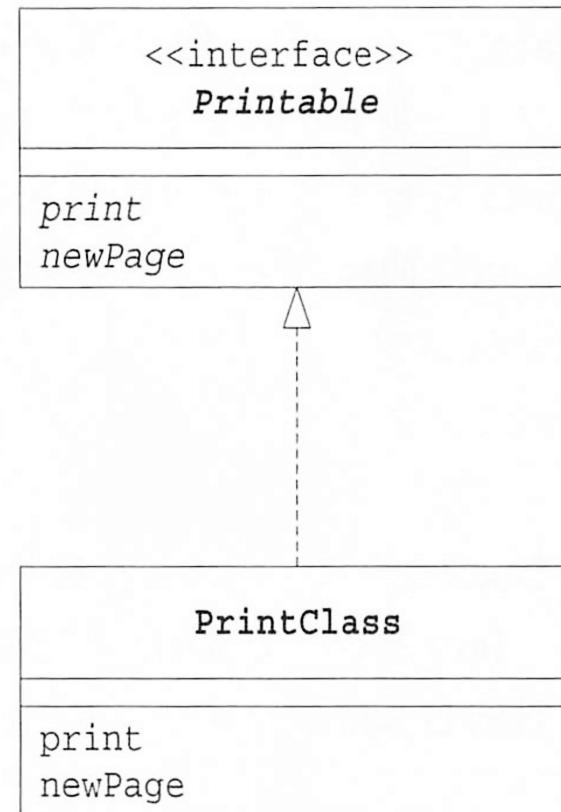


UMLによるクラス図 (2/4)

❖ インタフェースの実装

```
interface Printable {
    abstract void print();
    abstract void newPage();
}

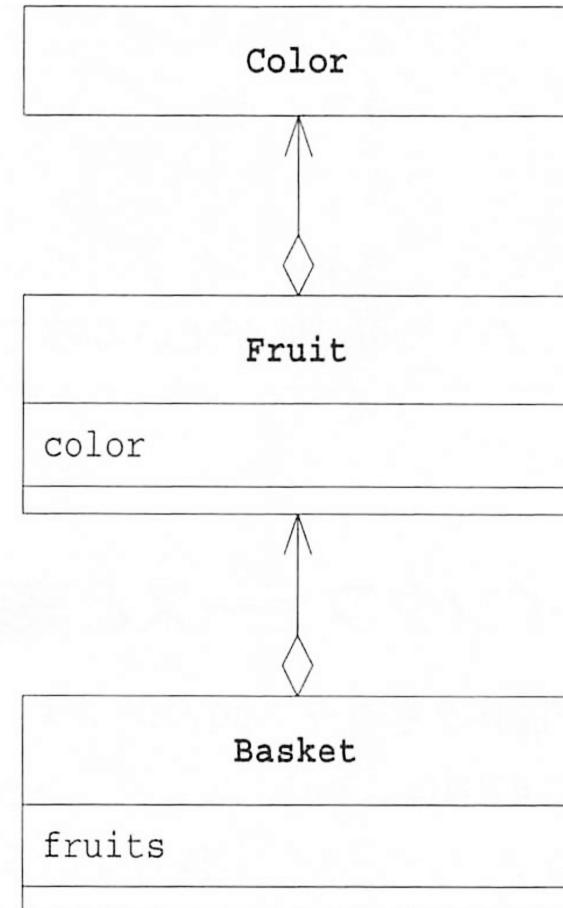
class PrintClass implements Printable {
    void print() {
        // ...
    }
    void newPage() {
        // ...
    }
}
```



UMLによるクラス図 (3/4)

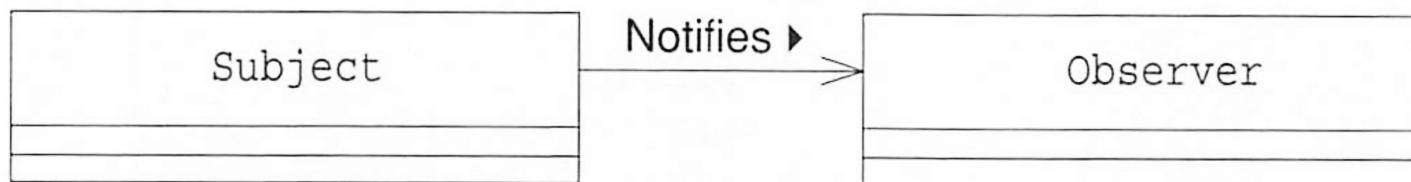
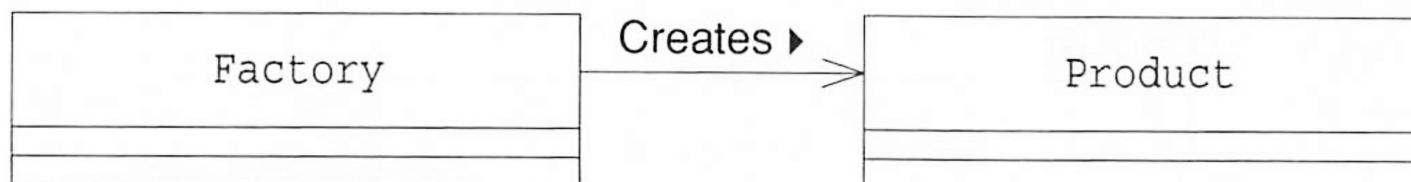
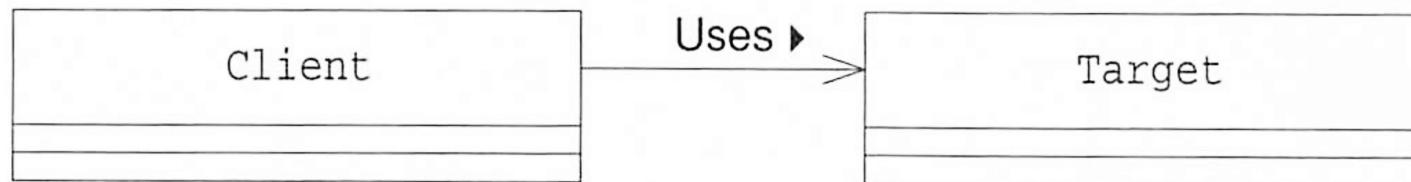
❖ 集約

```
class Color {  
    // ...  
}  
  
class Fruit {  
    Color color;  
    // ...  
}  
  
class Basket {  
    Fruit[] fruits;  
    // ...  
}
```



UMLによるクラス図 (4/4)

❖ その他のクラスの関係



Adapterパターン

日常生活におけるAdapter

- ❖ 必要なものと、提供されているものとの間に入って、その間を埋めるもの
- ❖ adapt：適応させる
- ❖ 例
 - ❖ 直流12V → 交流100V
 - ❖ USB-C → USB-A



デザインパターンにおけるAdapter

- ❖ 必要なコードと、 提供されているコードとの間に入って、 その間を埋めるコード
- ❖ 様々な理由で、 必要なコード・提供されているコードを変更できないことがある
 - ❖ 例：変更権限がない、 変更すると他のコードに影響が出る
- ❖ 典型的には必要なコードが「仕様」、 提供されているコードが「外部ライブラリ」

Adapterパターンの説明のための想定シーン

- ❖ 必要なもの（仕様）

- ❖ printWeak()で控えめに表示できること
- ❖ printStrong()で強調して表示できること

- ❖ 提供されているもの（外部ライブラリ）

- ❖ showWithParen()・showWithAster()を備えたBannerクラス

```
public void showWithParen() {  
    System.out.println("(" + string + ")");  
}  
  
public void showWithAster() {  
    System.out.println("*" + string + "*");  
}
```

- ❖ 外部ライブラリを活用し、仕様を満たすプログラムを作成するシーンを考える

- ❖ 当然ながら、仕様・外部ライブラリを変更することはできない
- ❖ つまり、仕様と外部ライブラリの溝を埋めるAdapterを作成する必要がある

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/02`
 - ❖ 以降、SOMEWHERE/2021_ap/02をWORK_DIRとする
- ❖ 「Adapterパターン IR02-1」の作業ディレクトリの作成・移動
 - ❖ `cd WORK_DIR`
 - ❖ Bb > 02: Adapter & Iterator > adapter_prac1.zip をWORK_DIRにダウンロード
 - ❖ `unzip adapter_prac1.zip`
 - ❖ `cd adapter_prac1`

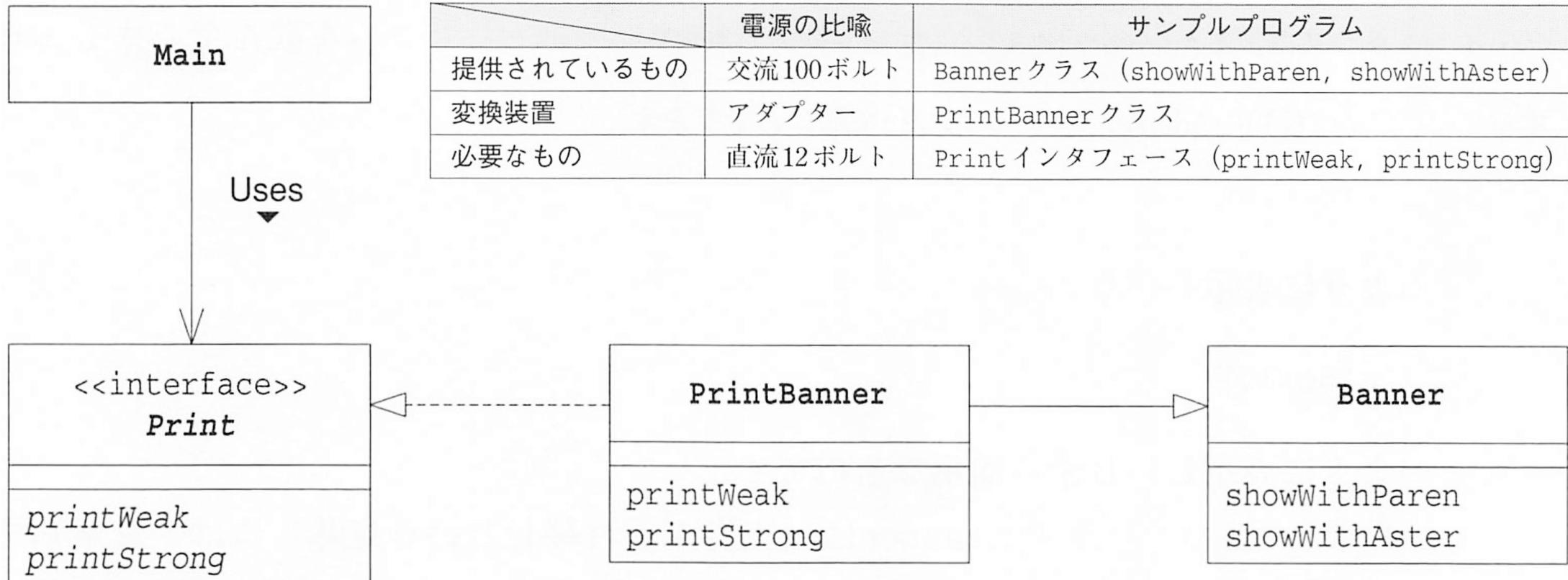
IR02-1

- ❖ adapter_prac1内に、Printインターフェースを実装するPrintBannerクラス（このクラスがAdapterの役割を果たす）を作成し、目標の結果が得られるプログラムを作成せよ。ただし、PrintBannerクラス以外は作成禁止、既存クラスは変更禁止とする。

目標の結果

```
(Hello)  
*Hello*
```

継承を用いたAdapterパターン (1/2)



継承を用いたAdapterパターン (2/2)

Main.java

```
public class Main {
    public static void main(String[] args) {
        Print print = new PrintBanner("Hello");
        print.printWeak();
        print.printStrong();
    }
}
```

Print.java (必要なもの) PrintBanner.java (Adapter)

```
public interface Print {
    void printWeak();
    void printStrong();
}
```

```
public class PrintBanner extends Banner implements Print {
    public PrintBanner(String string) {
        super(string);
    }

    public void printWeak() {
        showWithParen();
    }

    public void printStrong() {
        showWithAster();
    }
}
```

Banner.java (提供されたもの)

```
public class Banner {
    private String string;

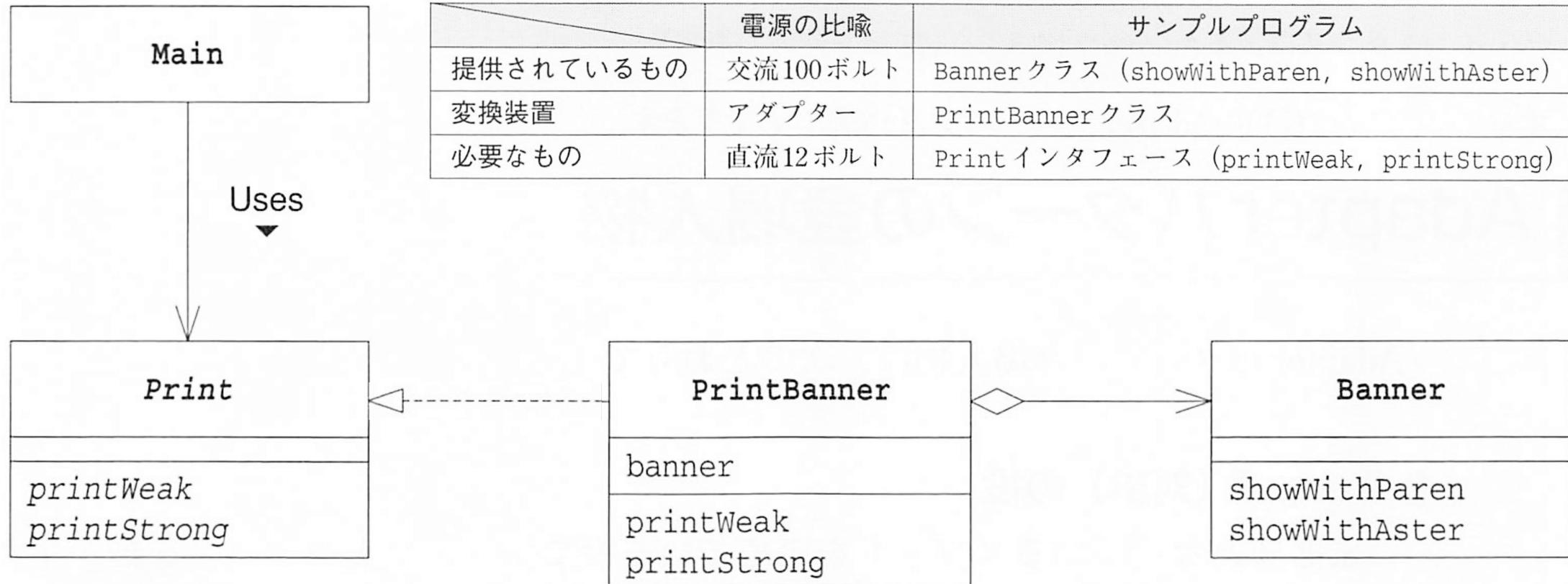
    public Banner(String string) {
        this.string = string;
    }

    public void showWithParen() {
        System.out.println("(" + string + ")");
    }

    public void showWithAster() {
        System.out.println("*" + string + "*");
    }
}
```

printWeak()をshowWithParen()に、
printStrong()をshowWithAster()に
読み替えている（適応させている）

移譲を用いたAdapterパターン (1/2)



移譲を用いたAdapterパターン (2/2)

Main.java

```
public class Main {
    public static void main(String[] args) {
        Print print = new PrintBanner("Hello");
        print.printWeak();
        print.printStrong();
    }
}
```

Print.java (必要なもの)

```
public interface Print {
    void printWeak();
    void printStrong();
}
```

PrintBanner.java (Adapter)

```
public class PrintBanner implements Print {
    private Banner banner;

    public PrintBanner(String string) {
        banner = new Banner(string);
    }

    public void printWeak() {
        banner.showWithParen();
    }

    public void printStrong() {
        banner.showWithAster();
    }
}
```

Banner.java (提供されたもの)

```
public class Banner {
    private String string;

    public Banner(String string) {
        this.string = string;
    }

    public void showWithParen() {
        System.out.println("(" + string + ")");
    }

    public void showWithAster() {
        System.out.println("*" + string + "*");
    }
}
```

printWeak()をshowWithParen()に、
printStrong()をshowWithAster()に
読み替えている（適応させている）

作業準備

- ❖ 「Adapterパターン IR02-2」の作業ディレクトリの作成・移動
 - ❖ `cd WORK_DIR`
 - ❖ Bb > 02: Adapter & Iterator > adapter_prac2.zip をWORK_DIRにダウンロード
 - ❖ `unzip adapter_prac2.zip`
 - ❖ `cd adapter_prac2`

IR02-2

- ❖ adapter_prac2内にて、必要最低限の新クラス作成と既存クラス修正を行い、目標の結果を得られるプログラムを作成せよ。

目標の結果（1行目がprintWeak(), 2～4行目がprintStrong()の出力である）

```
###Hello###  
####  
Hello  
####
```

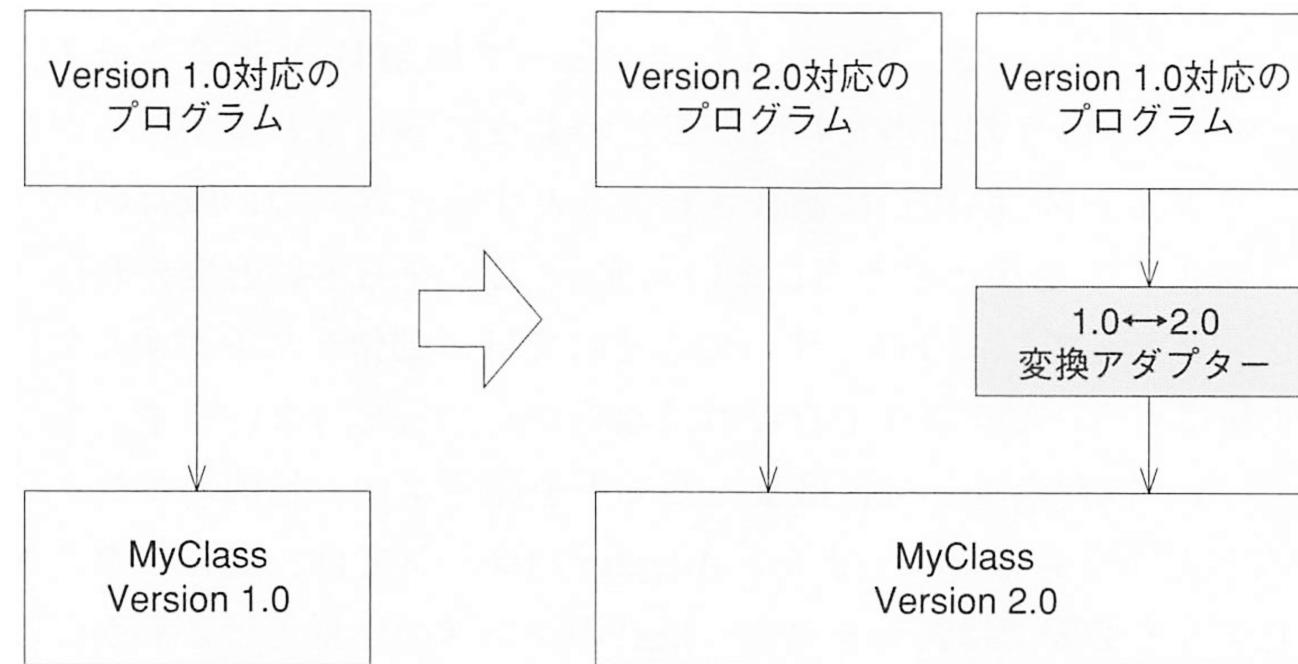
Adapterパターンのまとめと利用シーン (1/2)

- ❖ 必要なコードと、 提供されているコードとの間に入って、 その間を埋めるコードを書くという考え方
- ❖ 利用シーン1
 - ❖ 既に仕様が決まっており、 自分には仕様を変更する権限がない
 - ❖ 仕様と似たことが達成できそうだが、 微妙に挙動が異なる外部ライブラリがある
→ 仕様と外部ライブラリの溝を埋めるAdapterプログラムを書けば良い
- ❖ 利用シーン2
 - ❖ 既に徹底的にテストされた巨大な既存プログラムがあり、 できればそこに手を入れたくない
 - ❖ 既存プログラムに新機能を追加するライブラリがあるが、 既存プログラムのインターフェースと整合しておらず、 そのままライブラリを導入することはできない
→ 既存プログラムとライブラリの溝を埋めるAdapterプログラムを書けば良い
(仮にバグが生じても、 原因はAdapterかライブラリにあると絞り込める)

Adapterパターンのまとめと利用シーン (2/2)

◆利用シーン3

- ◆ システムのバージョンアップを行いたいが、それを行うと旧バージョンのブラウザをサポートできなくなる
- ◆ 旧バージョンのブラウザのコードを全部書き直すリソース（金・時間）は無い
→ 新システムと旧バージョンブラウザ関連コードの溝を埋めるAdapterプログラムを書けば良い



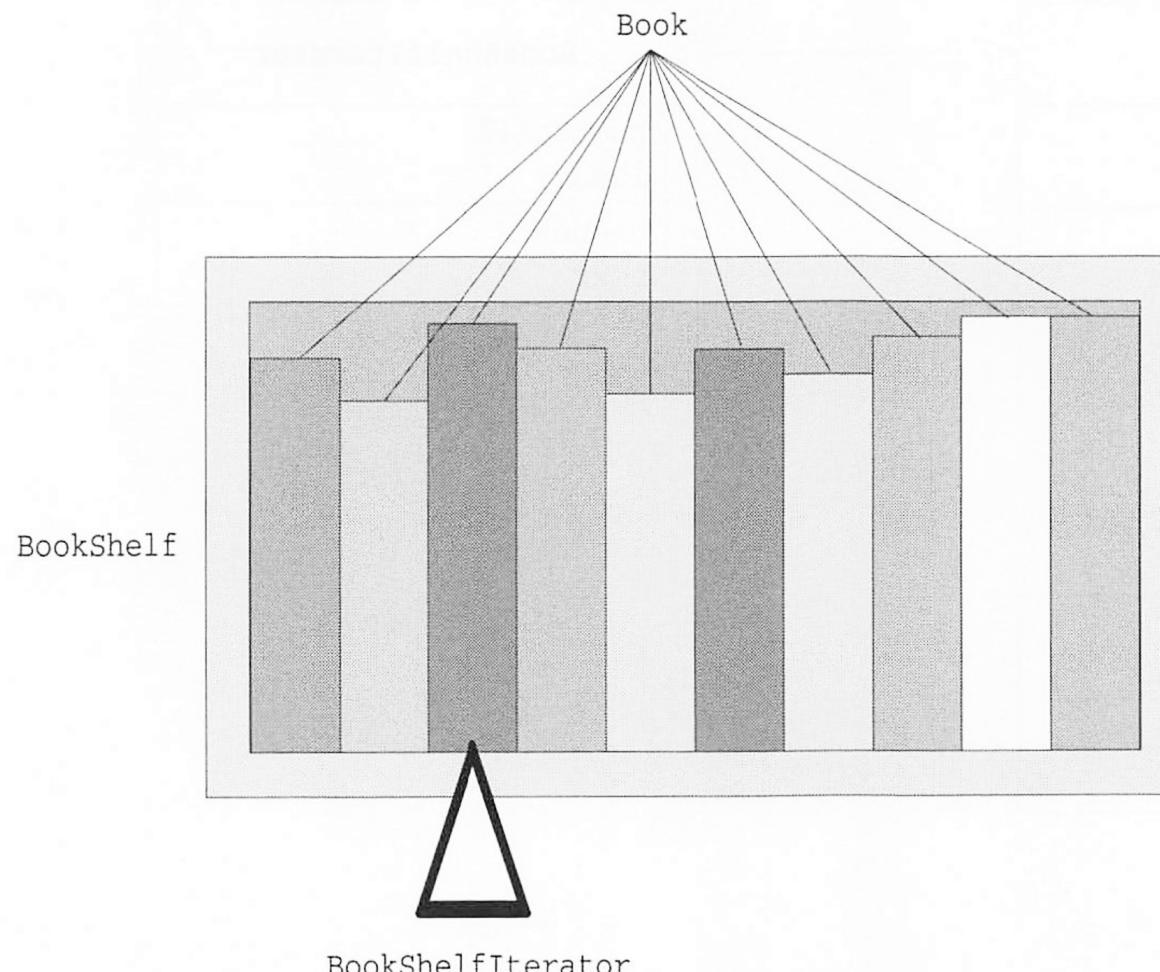
Iteratorパターン

Iteratorパターン

- ❖ 何かがたくさん集まっているとき、各要素を順番に指示していく処理を方法を統一するための考え方
- ❖ iterate：繰り返す
- ❖ for／whileで配列を走査する際のループ変数iを抽象化したものと捉えてもよい

Iteratorパターンの説明のための想定シーン

- ❖ Bookの集合を含むBookShelfを走査する



Iteratorパターンを用いない場合の問題点

- ❖ BookShelfの具体的な実装方法（例：配列、ArrayList）を把握しないとMainクラスが書けない
- ❖ BookShelfの実装方法が変わるとたびにMainクラスを修正しないといけない

Main.java (配列版)

```
public class Main {  
    public static void main(String[] args) {  
        Book[] bookShelf = new Book[4];  
        bookShelf[0] = new Book("Animal Farm");  
        bookShelf[1] = new Book("Bible");  
        bookShelf[2] = new Book("Cinderella");  
        bookShelf[3] = new Book("Daddy-Long-Legs");  
  
        for(int i = 0; i < bookShelf.length; i++) {  
            Book book = bookShelf[i];  
            System.out.println(book.getName());  
        }  
    }  
}
```

Main.java (ArrayList版)

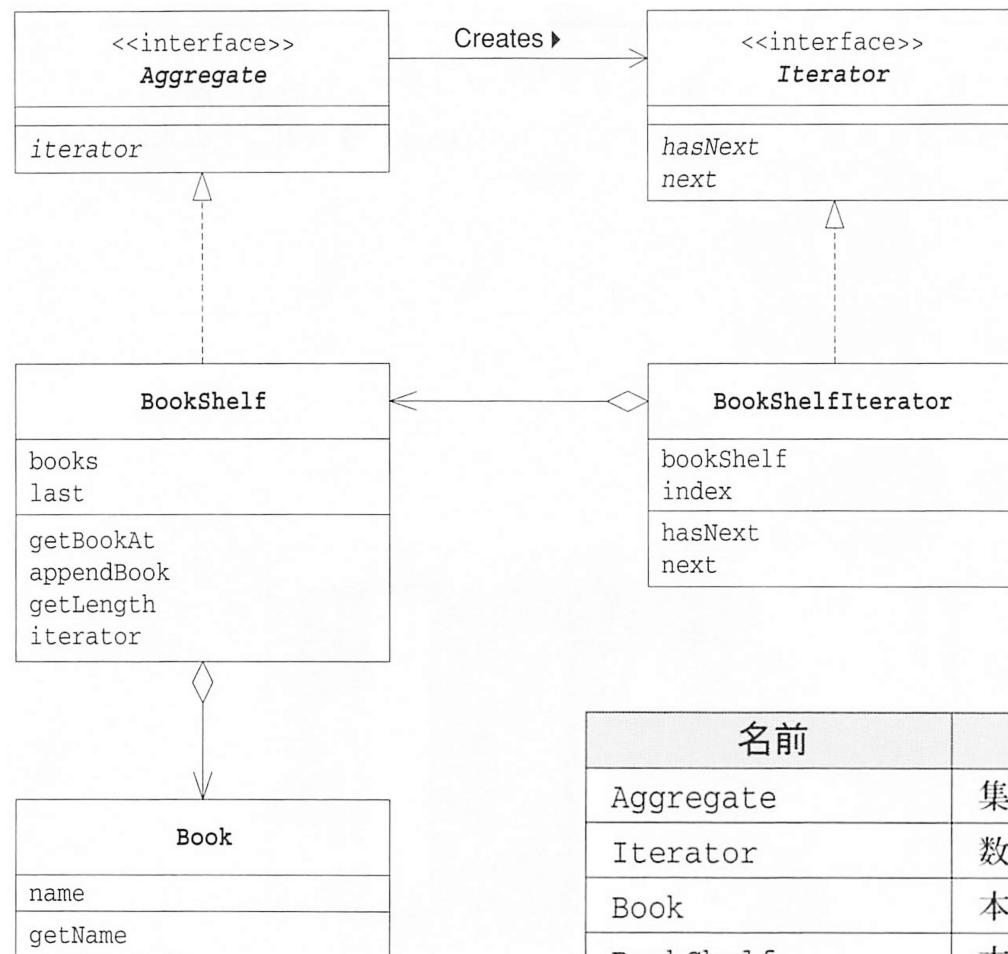
```
import java.util.ArrayList;  
  
public class Main2 {  
    public static void main(String[] args) {  
        ArrayList<Book> bookShelf = new ArrayList<Book>();  
        bookShelf.add(new Book("Animal Farm"));  
        bookShelf.add(new Book("Bible"));  
        bookShelf.add(new Book("Cinderella"));  
        bookShelf.add(new Book("Daddy-Long-Legs"));  
  
        for(int i = 0; i < bookShelf.size(); i++) {  
            Book book = (Book)bookShelf.get(i);  
            System.out.println(book.getName());  
        }  
    }  
}
```

Book.java

```
public class Book {  
    private String name;  
  
    public Book(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

bookShelfを配列で書く場合と
ArrayListで書く場合で
Mainクラスが大きく異なる

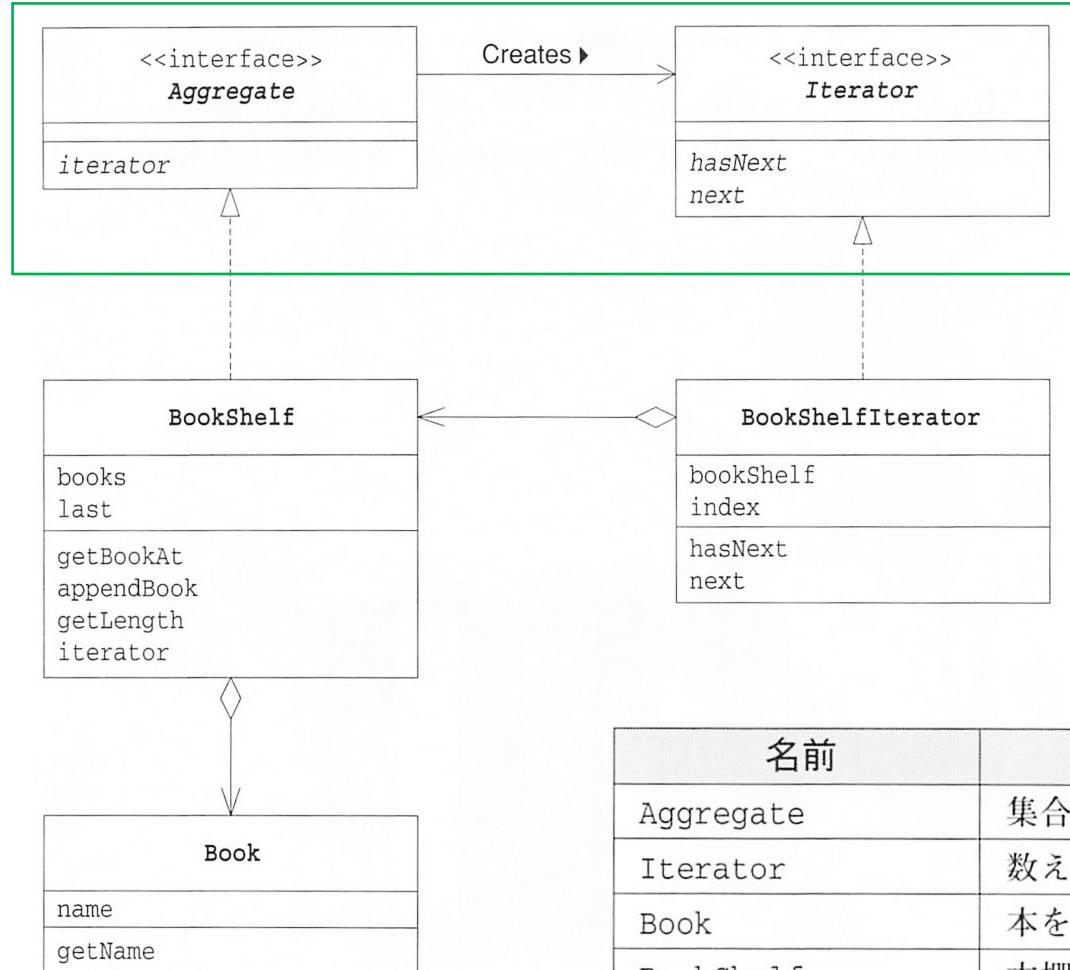
Iteratorパターン (1/5)



| 名前 | 解説 |
|-------------------|----------------------|
| Aggregate | 集合体を表すインターフェース |
| Iterator | 数え上げ、スキャンを行うインターフェース |
| Book | 本を表すクラス |
| BookShelf | 本棚を表すクラス |
| BookShelfIterator | 本棚をスキャンするクラス |
| Main | 動作テスト用のクラス |

Iteratorパターン (2/5)

Aggregate (集合体) は
数え上げの道具を
返却できる。

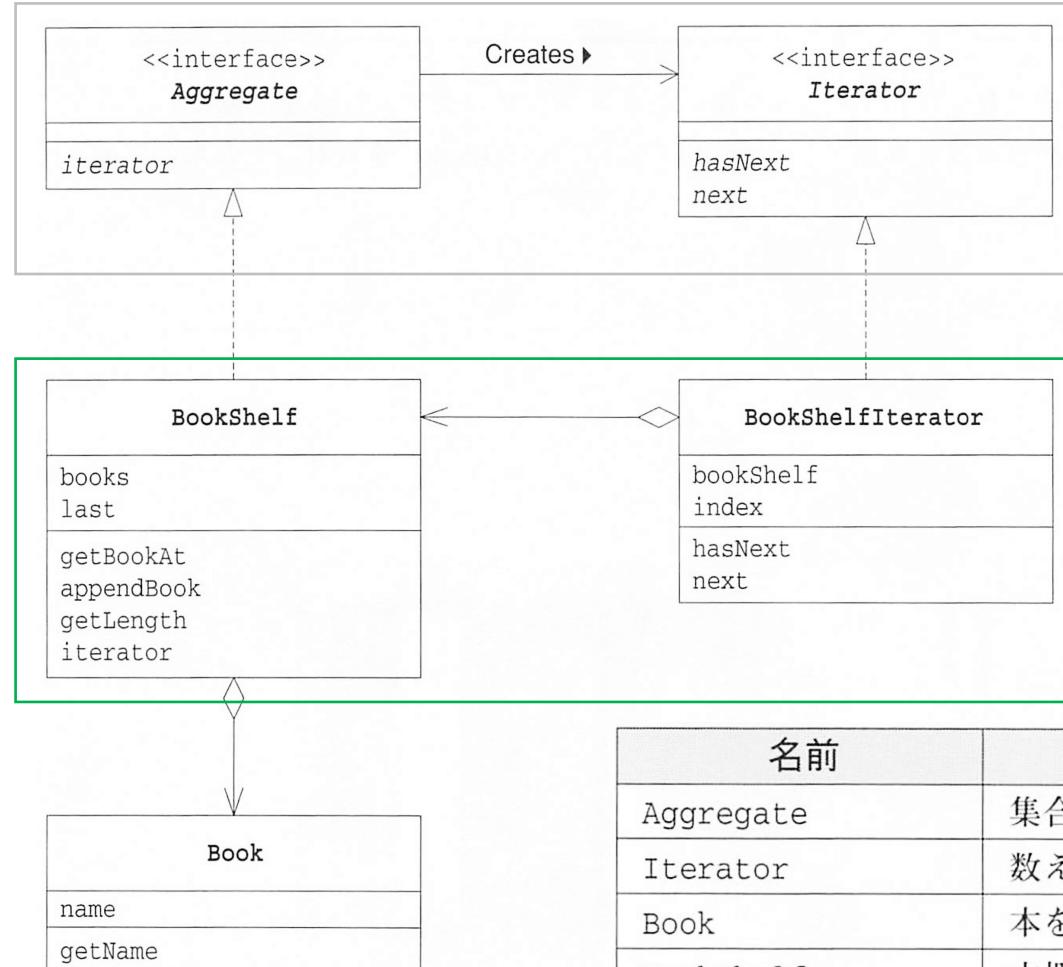


Iterator (数え上げの道具) は
集合体を走査して
次の要素の有無 (hasNext) と,
次の要素 (next) を返却できる

| 名前 | 解説 |
|-------------------|----------------------|
| Aggregate | 集合体を表すインターフェース |
| Iterator | 数え上げ、スキャンを行うインターフェース |
| Book | 本を表すクラス |
| BookShelf | 本棚を表すクラス |
| BookShelfIterator | 本棚をスキャンするクラス |
| Main | 動作テスト用のクラス |

Iteratorパターン (3/5)

Aggregate（集合体）は
数え上げの道具を
返却できる。



BookShelfクラスは
Aggregateインターフェースを
具体的に実装している

Iterator（数え上げの道具）は
集合体を走査して
次の要素の有無 (hasNext) と,
次の要素 (next) を取得できる

BookShelfIteratorクラスは
Iteratorインターフェースを
具体的に実装している

| 名前 | 解説 |
|-------------------|----------------------|
| Aggregate | 集合体を表すインターフェース |
| Iterator | 数え上げ、スキャンを行うインターフェース |
| Book | 本を表すクラス |
| BookShelf | 本棚を表すクラス |
| BookShelfIterator | 本棚をスキャンするクラス |
| Main | 動作テスト用のクラス |

Iteratorパターン

(4/5)

Aggregate.java

```
public interface Aggregate {  
    Iterator iterator();  
}
```

BookShelf.java

```
public class BookShelf implements Aggregate {  
    private Book[] books;  
    private int last = 0;  
  
    public BookShelf(int maxSize) {  
        books = new Book[maxSize];  
    }  
  
    public Book getBookAt(int index) {  
        return books[index];  
    }  
  
    public void appendBook(Book book) {  
        books[last] = book;  
        last++;  
    }  
  
    public int getLength() {  
        return last;  
    }  
  
    public Iterator iterator() {  
        return new BookShelfIterator(this);  
    }  
}
```

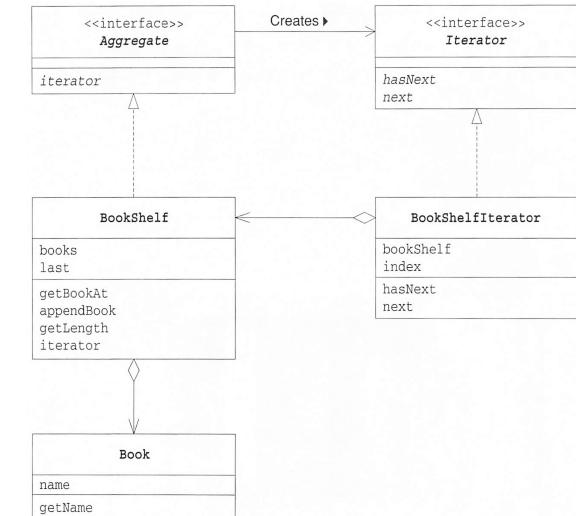
Iterator.java

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
}
```

BookShelfIterator.java

```
public class BookShelfIterator implements Iterator {  
    private BookShelf bookShelf;  
    private int index;  
  
    public BookShelfIterator(BookShelf bookShelf) {  
        this.bookShelf = bookShelf;  
        index = 0;  
    }  
  
    public boolean hasNext() {  
        if(index < bookShelf.getLength()) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    public Object next() {  
        Book book = bookShelf.getBookAt(index);  
        index++;  
        return book;  
    }  
}
```

これらのインターフェースでは
BookShelfの具体的な実装を
意識していない点に注目



Iteratorパターン (5/5)

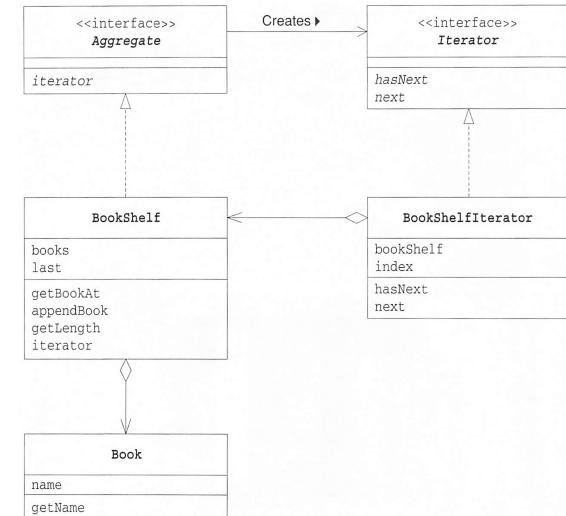
Book.java

```
public class Book {  
    private String name;  
  
    public Book(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

Main.java

```
public class Main {  
    public static void main(String[] args) {  
        BookShelf bookShelf = new BookShelf(4);  
        bookShelf.appendBook(new Book("Animal Farm"));  
        bookShelf.appendBook(new Book("Bible"));  
        bookShelf.appendBook(new Book("Cinderella"));  
        bookShelf.appendBook(new Book("Daddy-Long-Legs"));  
  
        Iterator it = bookShelf.iterator();  
        while(it.hasNext()) {  
            Book book = (Book)it.next();  
            System.out.println(book.getName());  
        }  
    }  
}
```

BookShelfの具体的な実装を
意識していない走査方法



作業準備

- ❖ 「Iteratorパターン IR02-3」の作業ディレクトリの作成・移動
 - ❖ `cd WORK_DIR`
 - ❖ `Bb > 02: Adapter & Iterator > iterator_prac.zip` をWORK_DIRにダウンロード
 - ❖ `unzip iterator_prac.zip`
 - ❖ `cd iterator_prac`

IR02-3

❖ iterator_prac内にて、必要最低限の修正を行い、Bookを配列ではなくArrayListで管理できるようにせよ。

Class ArrayList<E>

```
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>
```

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

ArrayListの使用方法は
P.38や公式API等のWebページを参照のこと

Iteratorパターンのまとめと利用シーン

- ❖ 何かがたくさん集まっているとき、各要素を順番に指示していく処理を方法を統一するための考え方
- ❖ 利用シーン
 - ❖ 集合体を走査したいが、集合体の表現が変わる可能性がある場合
 - ❖ 例えば、配列かもしれないし、ArrayListかもしれないし、未知のデータ構造かもしれない

本日のまとめ

❖ 講義内容

- ❖ デザインパターンとは
- ❖ Adapterパターン
- ❖ Iteratorパターン

❖ 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。