

Javaプログラミング2

オブジェクト指向の基礎

目次

➤ 第1章 クラスの基本.....5	➤ 第5章 継承.....73
➤ クラスとインスタンス.....6	➤ 継承の基本.....74
➤ コンストラクタ.....17	➤ メソッドのオーバーライド.....86
➤ 参照型変数.....24	➤ 継承とコンストラクタ.....94
➤ クラスと配列.....30	➤ その他のトピックス.....98
➤ 第2章 オーバーロード.....33	➤ 第6章 ポリモーフィズム.....103
➤ メソッドのオーバーロード.....34	➤ クラスの型変換.....104
➤ コンストラクタのオーバーロード.....41	➤ ポリモーフィズムの基本.....108
➤ 第3章 static修飾子.....47	➤ 抽象クラス.....114
➤ static修飾子とは.....48	➤ インターフェイス.....120
➤ 第4章 カプセル化.....55	➤ 第7章 パッケージ.....129
➤ アクセス修飾子.....57	➤ パッケージ.....130
➤ カプセル化.....65	➤ クラスパス.....146
	➤ JARファイル.....151

目次(つづき)

- 第8章 例外処理.....155
 - 例外の基本.....156
 - try-catch-finallyによる例外処理....165
 - throwsによる例外処理.....173
 - 例外クラスの構造.....177
 - Java7の新機能.....183
 - その他のトピックス.....186
- 第9章 オブジェクト指向プログラミングの
 ポイント.....191
 - 保守性と再利用性.....192
 - 単一責任の原則.....193

開発環境など

- ▶ このテキストでは、以下の開発環境を前提としています。
 - ▶ Windows 7(64bit版)
 - ▶ JDK 8

第1章

クラスの基本

クラスとインスタンス

クラスの定義

▼ 社員クラス (Employee) の定義

```
class Employee {  
  {  
    int id;      クラス名  
    String name;  
  }  
  {  
    void introduce() {  
      System.out.println("社員番号=" + id);  
      System.out.println("氏名=" + name);  
    }  
  }  
}
```

フィールド

メソッド

フィールド・メソッド

➡ フィールド

- ➡ クラスに属する変数
 - ➡ メソッドの外で定義する

➡ メソッド

- ➡ クラスに属する処理
 - ➡ なぜstaticが付いていないかは後ほど説明

➡ フィールドとメソッドをあわせて、クラスの「メンバ」とも言う

クラスの利用

Employeeクラスのインスタンスを生成する

```
class EmployeeMain {  
    public static void main(String[] args) {  
        Employee emp1 = new Employee();  
        Employee emp2 = new Employee();  
        emp1.id = 10001;  
        emp1.name = "内田太郎";  
        emp2.id = 10002;  
        emp2.name = "鈴木花子";  
        emp1.introduce();  
        emp2.introduce();  
    }  
}
```

インスタンス生成、
変数への代入

フィールドへの値の代入

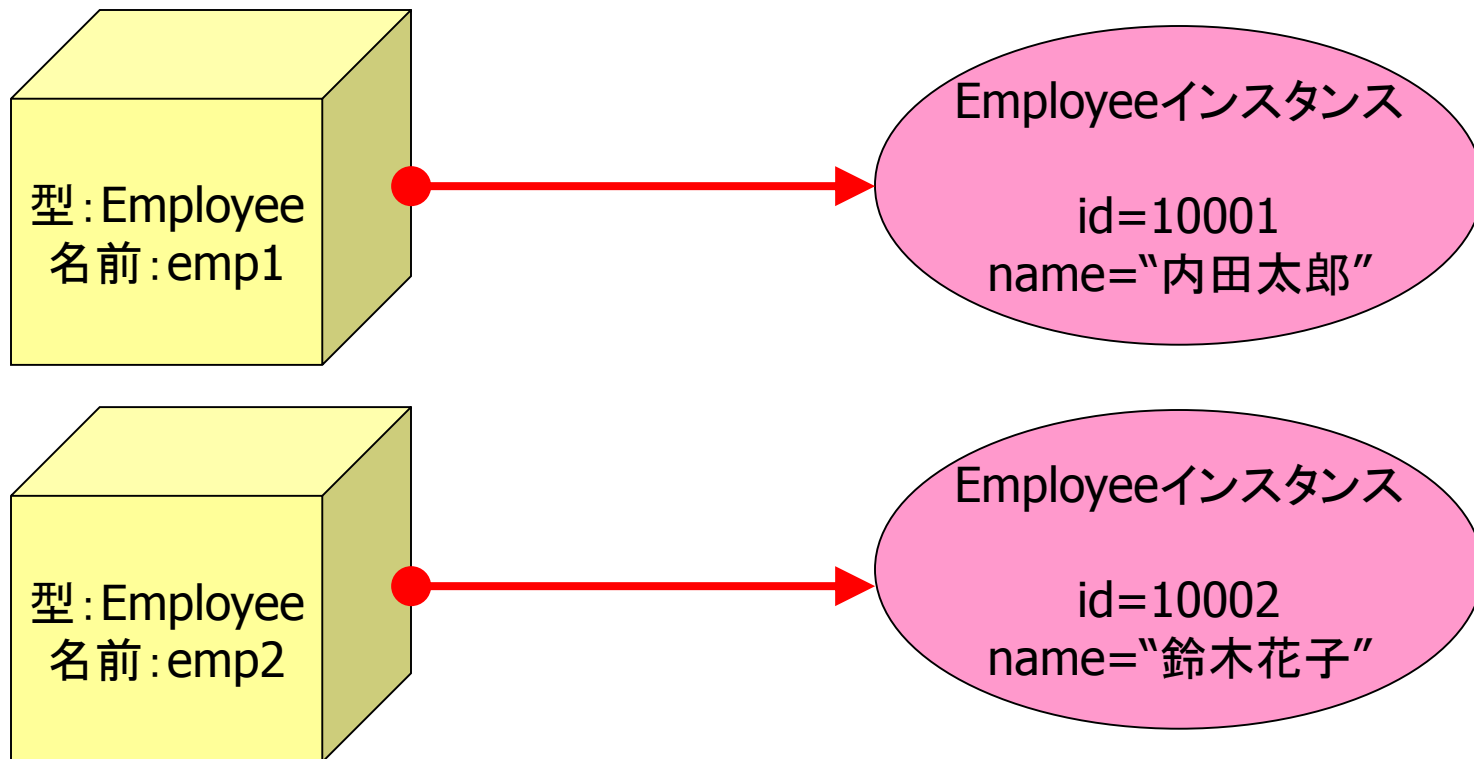
メソッドの呼び出し

実行結果

```
C:\¥java>java EmployeeMain  
社員番号=10001  
氏名=内田太郎  
社員番号=10002  
氏名=鈴木花子
```

インスタンスの生成

- ▶ 「**new クラス名()**」でインスタンスを生成する
- ▶ インスタンスを生成することを「**newする**」とも言う。



クラスとインスタンスの関係

クラス	インスタンス
社長	孫正義、三木谷浩史、スティーブ・ジョブズ...
野球選手	藤川球児、ラミレス、長嶋茂雄...
ボールペン	「この」ボールペン、「あの」ボールペン...

クラスとインスタンスの関係(続き)

▶ クラス = 定義

- ▶ Employeeクラスでは、「社員(Employee)とは、社員番号(id)と名前(name)を持っていて、自己紹介をする(introduce())もの」と定義している

▶ インスタンス = 実体(実際に存在しているもの)

- ▶ 具体的な社員(Employee)の実体(内田太郎、鈴木花子...)
- ▶ 「オブジェクト」という場合もある

フィールドへのアクセス

- ▶ <インスタンス変数名>.<フィールド名>でフィールドにアクセスできる

```
Employee e = new Employee();  
e.name = "田中"; // フィールドへの代入  
String n = e.name; // フィールドの値の取得  
System.out.println(n); // 「田中」と表示される
```

メソッドの呼び出し

- ▶ <インスタンス変数名>.<メソッド名>()でメソッドを呼び出すことが出来る

```
Employee e = new Employee();  
e.id = 10000;  
e.name = "田中";  
e.introduce(); // メソッド呼び出し
```

フィールドとローカル変数のスコープ

- ▶ フィールド＝クラスに属する変数
 - ▶ クラス内のどこからでも参照できる
 - ▶ クラス外からでもアクセスできる(出来ない場合もある)
- ▶ ローカル変数＝メソッド内で宣言された変数
 - ▶ そのメソッド内でしか参照できない

```
class Hoge { フィールドのスコープ
```

```
    String text;
```

```
    int num;
```

```
    void foo() { ローカル変数aのスコープ
```

```
        int a;
```

```
        // ...
```

```
    }
```

```
    void bar() { ローカル変数bのスコープ
```

```
        String b;
```


```
        // ...
```

```
    }
```

```
}
```

フィールドの初期値

- ▶ インスタンス生成時点で、各フィールドには「デフォルトの初期値」が代入される
 - ▶ boolean→false
 - ▶ 整数・小数→0
 - ▶ 参照型 (Stringなど)→null



コンストラクタ

コンストラクタとは

- ▶ インスタンス生成時に1回だけ実行される処理
- ▶ フィールドの初期化のために使うことが多い

```
class EmployeeCS {  
    int id;  
    String name;    コンストラクタ  
    EmployeeCS(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
    void introduce() {  
        System.out.println("社員番号=" + id);  
        System.out.println("氏名=" + name);  
    }  
}
```

thisキーワード

- ▶ 「このインスタンス」を表す
- ▶ 引数付きのコンストラクタでは、右記のような書き方をすることが多い

```
class EmployeeCS {  
    int id;  
    String name;  
    EmployeeCS(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
    ...  
}
```

コンストラクタの実行タイミング

▶ インスタンス生成時

- ▶ 複数のインスタンスを生成した場合、1つ1つのインスタンス生成時に毎回実行される

```
class EmployeeCSMain {  
    public static void main(String[] args) {  
        EmployeeCS emp1  
            = new EmployeeCS(10001, "内田太郎"); ← コンストラクタ実行  
        EmployeeCS emp2  
            = new EmployeeCS(10002, "鈴木花子"); ← コンストラクタ実行  
        emp1.introduce();  
        emp2.introduce();  
    }  
}
```

実行結果

```
C:\¥java>java EmployeeCSMain  
社員番号=10001  
氏名=内田太郎  
社員番号=10002  
氏名=鈴木花子
```

デフォルトコンストラクタ

- クラス内にコンストラクタを定義していない場合、コンパイラが自動的に引数なし・処理なしのコンストラクタ(=デフォルトコンストラクタ)を追加する

```
class Foo {  
}
```

||

```
class Foo {  
    public Foo() {  
    }  
}
```

コンパイラが自動で
追加する

デフォルトコンストラクタの注意点

- ▼ クラス内に1つでもコンストラクタを定義している場合、コンパイラはデフォルトコンストラクタの自動追加を行わない

```
class Foo {  
    Foo(int num) {  
        // 処理  
    }  
}
```

デフォルトコンストラクタは追加されない

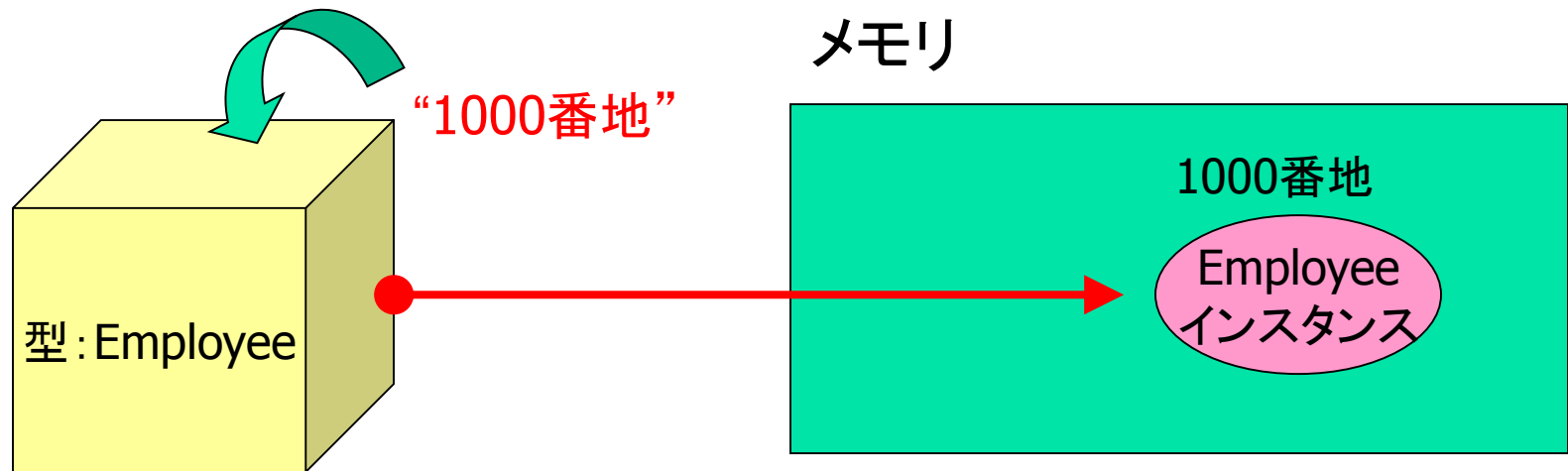
演習：自作クラス作成

- ▶ 自分で考えたクラスと、それを利用するクラス（mainメソッドあり）を作成する。
 - ▶ そのクラスには、どんなフィールド・メソッドが必要か？
- ▶ 例：Dog（犬）クラス
 - ▶ フィールド：体長、体重、名前、犬種・・・
 - ▶ メソッド：吠える、走る、情報の表示・・・

参照型変数

参照型変数

- ▶ 変数内にはインスタンスそのものではなく、インスタンスが存在する場所を表す値(参照値)が入っている
- ▶ 何も指していない場合、変数にはnullが入る



NullPointerException

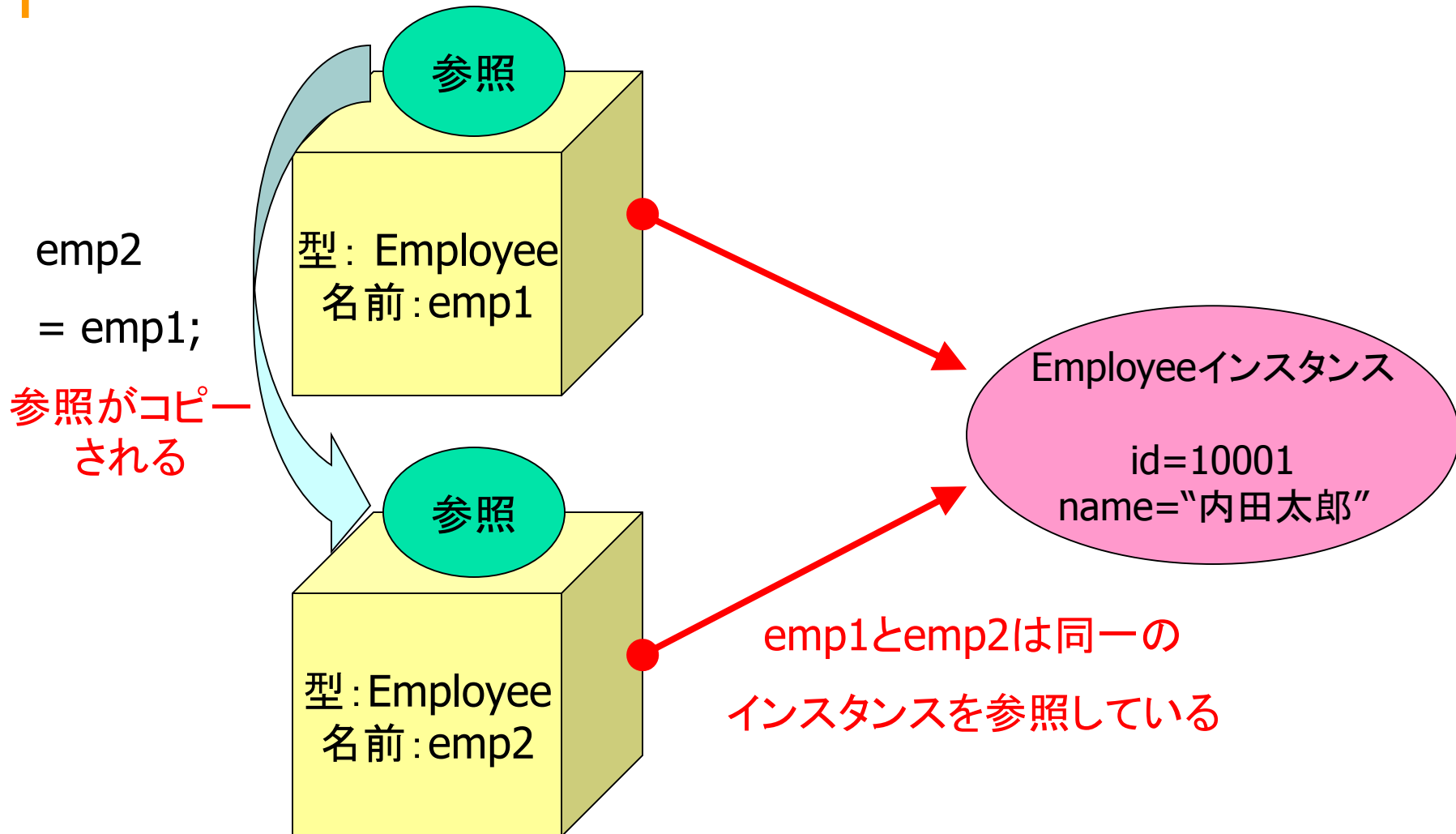
- ▶ インスタンス変数がnull(変数がインスタンスを参照していない)のときにフィールド・メソッドにアクセスしようとするが発生する例外

```
Employee emp = null;  
emp.introduce();
```

実行結果

```
Exception in thread "main" java.lang.NullPointerException  
at Foo.main(Foo.java:4)
```

参照型変数の代入



サンプルプログラム

```
class EmployeeRefMain {  
    public static void main(String[] args) {  
        Employee emp1 = new Employee();  
        Employee emp2 = emp1;  
        emp1.id = 10001;  
        emp1.name = "内田太郎";  
        emp1.introduce();  
        emp2.introduce();  
    }  
}
```

実行結果

```
C:\¥java>java EmployeeRefMain  
社員番号=10001  
氏名=内田太郎  
社員番号=10001  
氏名=内田太郎
```

emp1.introduce()とemp2.introduce()の実行結果が同じ

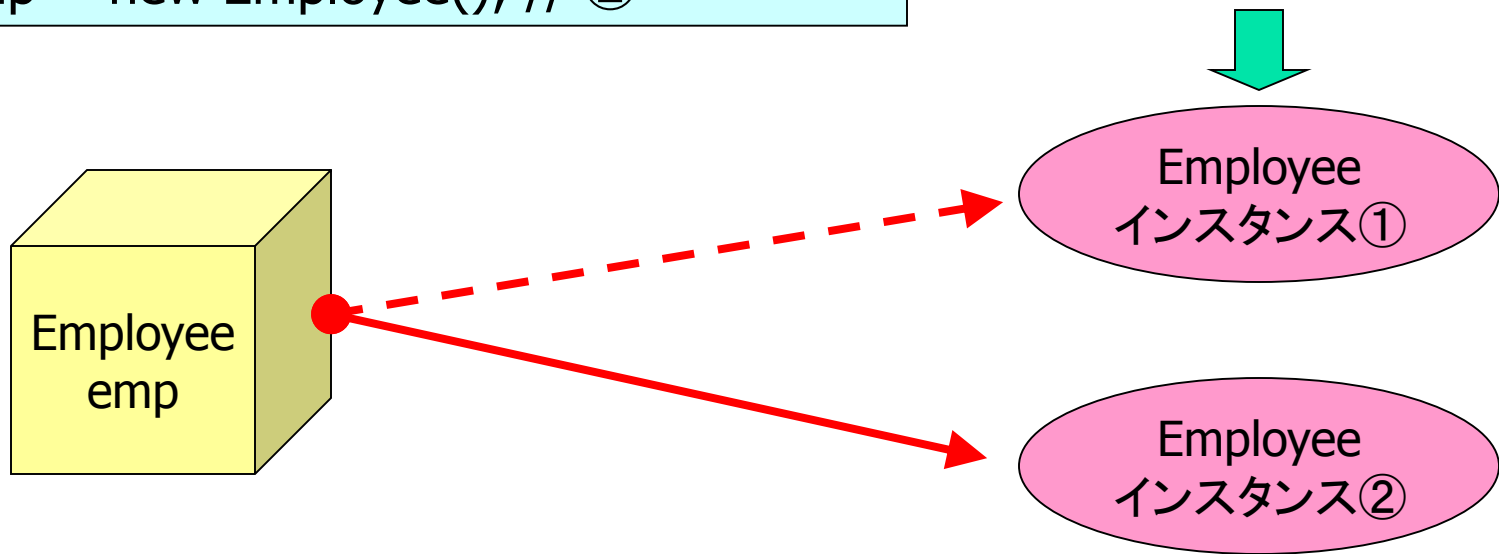
→emp1とemp2が同一のインスタンスを参照しているから

ガベージコレクション(ごみ集め・GC)

- ▶ 参照されなくなったインスタンスなどを、JVMが自動的にメモリから破棄すること

```
Employee emp = new Employee(); // ①  
emp = new Employee(); // ②
```

参照されなくなったインスタンスは、GCの対象となる



クラスと配列

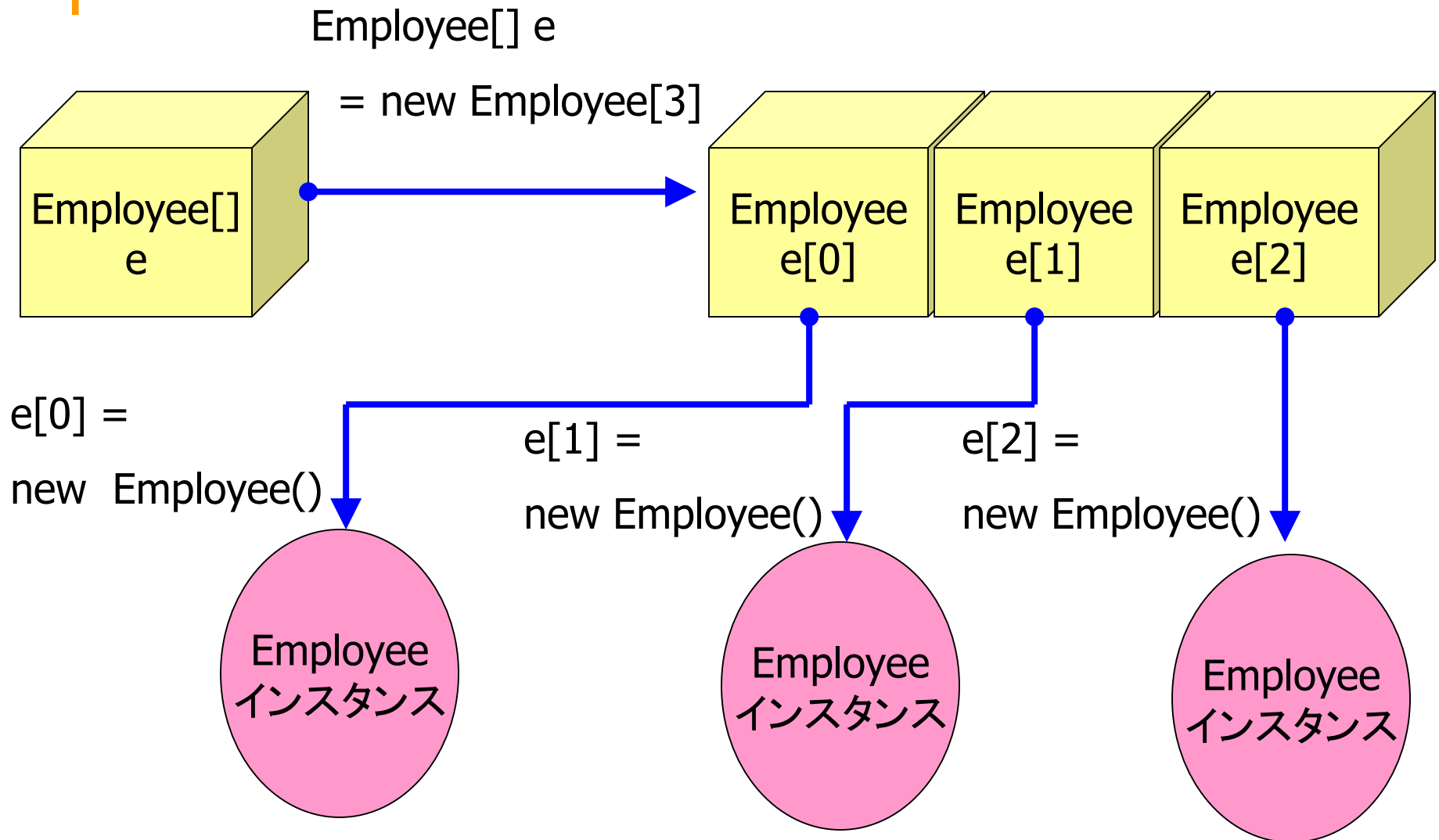
クラスと配列

```
Employee[] e = new Employee[3];  
e[0] = new Employee();  
e[1] = new Employee();  
e[2] = new Employee();
```

} 変数を3つ定義しただけ
→ e[0]~e[2]は全てnull

} インスタンスを生成し、
各変数に代入

クラス配列のイメージ図



第2章 オーバーロード

メソッドのオーバーロード

オーバーロードとは

- ▶ 同一クラス内に、同じ名前のメソッドを複数定義すること(=多重定義)

```
class Foo {  
    void method() { ... }  
    void method(int num) { ... }  
    void method(String str) { ... }  
}
```

サンプルプログラム

```
class Calculator {  
    int add(int num1, int num2) {  
        return num1 + num2;  
    }  
    double add(double num1, double num2) {  
        return num1 + num2;  
    }  
}
```

サンプルプログラム

```
class CalculatorMain {  
    public static void main(String[] args) {  
        Calculator cal = new Calculator();  
        int isum = cal.add(3, 5);  
        System.out.println("整数の和=" + isum);  
        double dsum = cal.add(3.14, 5.27);  
        System.out.println("小数の和=" + dsum);  
    }  
}
```

どちらのaddメソッドを実行するかは、引数の型を見てコンパイラが判断する

実行結果

```
C:\¥java>java CalculatorMain  
整数の和=8  
小数の和=8.41
```

オーバーロードのメリット

➡ 引数の型が違っていても、意識することなくメソッドを利用できる

➡ 例: `PrintStream#println()`

- ➡ `void println(byte b)`
- ➡ `void println(short s)`
- ➡ `void println(int i)`
- ➡ `void println(long l)`
- ➡ `void println(boolean b)`
- ➡ `void println(char c)`
- ➡ `void println(float f)`
- ➡ `void println(double d)`
- ➡ `void println(String s)`
- ➡ `void println(Object o)`
- ➡ `void println(char[] c)`
- ➡ `void println()`

メソッドのシグネチャ

- ◆ シグネチャ = メソッド名 + 引数の型の並び
 - ◆ 引数の名前は関係ない
- ◆ シグネチャが違えば、メソッド名が同じメソッドを同一クラス内に何個でも定義できる
 - ◆ 戻り値の型が違っていても、シグネチャが同一なら定義できない

シグネチャの例

- ▶ 違うシグネチャなので、同一クラス内で定義できる

```
void add(int a, int b) {}  
void add(double a, double b) {}
```

```
void add(int a) {}  
void add(int a, int b) {}
```

```
void method(int i, String s) {}  
void method(String s, int i) {}
```

- ▶ 同じシグネチャなので、同一クラス内で定義できない

```
void add(int a, int b) {}  
void add(int num1, int num2) {}
```

```
int add(int a, int b) {  
    return a + b;  
}  
double add(int a, int b) {  
    return a + b;  
}
```


コンストラクタのオーバーロード

コンストラクタのオーバーロード

▼ コンストラクタもオーバーロード可能

- ▼ 引数を変えることにより、コンストラクタを複数定義できる

```
class Foo {  
    Foo() { ... }  
    Foo(int num) { ... }  
    Foo(String str) { ... }  
}
```

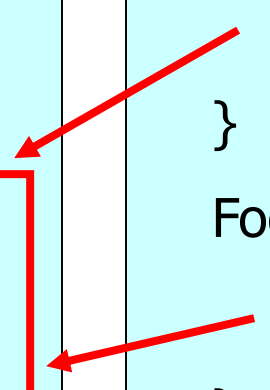
コンストラクタのthis()

そのクラスのコンストラクタを表す

```
class Foo {  
    String str;  
    int num;  
  
    Foo(String str, int num) {  
        this.str = str;  
        this.num = num;  
    }  
}
```

// 右に続く

```
Foo(String str) {  
    this(str, 999);  
}  
  
Foo() {  
    this("bar", 99);  
}  
...  
}
```



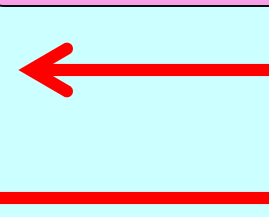
サンプルプログラム

```
class EmployeeOL {  
    int id;  
    String name;  
    EmployeeOL(int id, String name) {  
        this.id = id; this.name = name;  
    }  
    EmployeeOL(int id) {  
        this(id, "未入力");  
    }  
    void introduce() {  
        System.out.println("社員番号=" + id);  
        System.out.println("氏名=" + name);  
    }  
}
```

サンプルプログラム

```
class EmployeeOLMain {  
    public static void main(String[] args) {  
        EmployeeOL emp1  
            = new EmployeeOL(10001, "内田太郎");  
        EmployeeOL emp2  
            = new EmployeeOL(10002);  
        emp1.introduce();  
        emp2.introduce();  
    }  
}
```

どちらのコンストラクタを実行するかは、引数の型を見てコンパイラが判断する



実行結果

```
C:¥java>java EmployeeOLMain  
社員番号=10001  
氏名=内田太郎  
社員番号=10002  
氏名=未入力
```



第3章

static修飾子

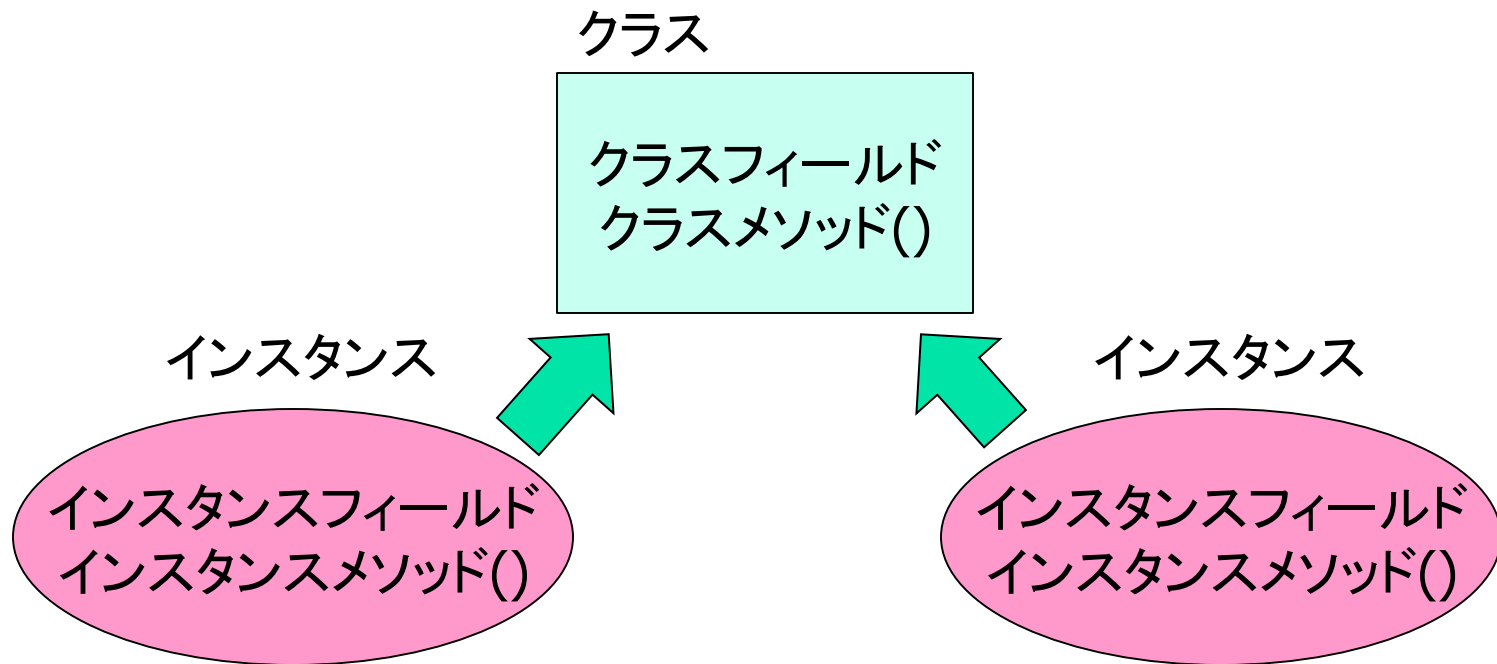
static修飾子とは

static修飾子とは

- ▶ static修飾子が付いたメソッドやフィールドは、インスタンスを生成しないで利用できる
 - ▶ クラスフィールド、クラスメソッドと呼ぶこともある
 - ▶ 利用方法
 - クラス名.フィールド名
 - クラス名.メソッド名()

イメージ図

- ▶ インスタンスメンバは、各インスタンスが持っている
- ▶ クラスメンバは、クラスが持っている
 - ▶ 各インスタンス間で共有されている



サンプルプログラム

```
class StaticSample {  
    int iField;  
    static int sField;  
    void iMethod() {  
        System.out.println("iField = " + iField);  
    }  
    static void sMethod() {  
        System.out.println("sField = " + sField);  
    }  
}
```

サンプルプログラム

```
class StaticSampleMain {  
    public static void main(String[] args) {  
        StaticSample.sField = 999;  
        StaticSample.sMethod();  
        StaticSample s = new StaticSample();  
        s.iField = 111;  
        s.iMethod();  
    }  
}
```

実行結果

```
C:¥java>java StaticSampleMain  
sField = 999  
iField = 111
```

staticフィールドの用途

- ▶ static finalな定数として使うことが多い
 - ▶ マジックナンバーの使用を防ぐ

```
public class Foo {  
    static final int TAX_RATE = 5; // 消費税率  
    ...  
}
```

staticメソッドの用途

- ▶ メソッドをstaticにする場合
 - ▶ メソッドを簡単に呼び出したい場合など
- ▶ 多用は避ける
 - ▶ staticにせざるを得ないメソッドのみstaticにすべき

第4章 カプセル化

オブジェクト指向の3大機能

- ▶ カプセル化（隠蔽）
- ▶ 継承（インヘリタンス）
- ▶ ポリモーフィズム（多態性、多相性）

アクセス修飾子

アクセス修飾子とは

- ▶ クラスの外部(=他のクラス)からのアクセスを制御する(=**利用を制限する**)ための修飾子
- ▶ クラス、フィールド、メソッド、コンストラクタなどに付ける

```
public class Foo {  
    private int field;  
    public Foo() { ... }  
    protected void method() { ... }  
}
```

アクセス修飾子の種類

広

アクセス可能範囲

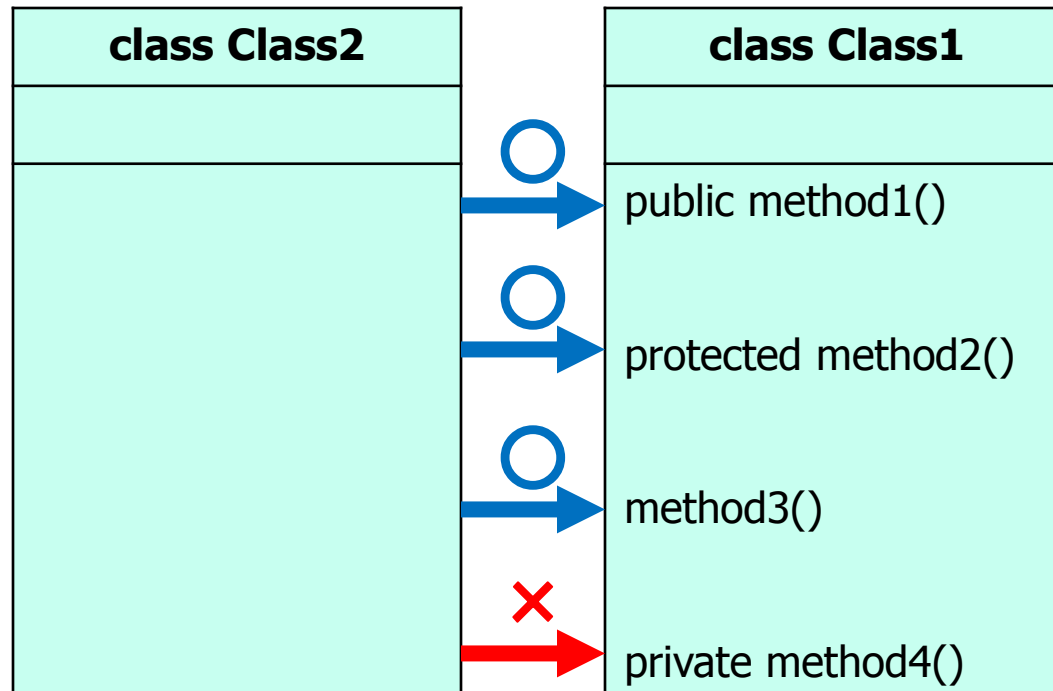
狭

アクセス修飾子	説明
public	パッケージ等に関わらず、すべてのクラスからアクセスできる
protected	同一パッケージ内のクラス、またはサブクラスからアクセスできる
指定なし(デフォルト、パッケージプライベート)	同一パッケージ内のクラスからのみアクセスできる
private	同じクラス内からのみアクセスできる

アクセス修飾子とメンバの利用

○・・・利用可能

×・・・利用不可



サンプルプログラム

```
class Class1 {  
    public void method1() {}  
    protected void method2() {}  
    void method3() {}  
    private void method4() {}  
}
```

```
class Class2 {  
    public static void main(String[] args) {  
        Class1 c = new Class1();  
        c.method1(); // ○  
        c.method2(); // ○  
        c.method3(); // ○  
        c.method4(); // × コンパイルエラー  
    }  
}
```

privateメソッドの用途

- ▶ 同一クラス内の他のメソッドから利用する
 - ▶ 他のメソッドから一部の処理を切り出す
 - ▶ 複数のメソッドから利用される共通機能を提供する

```
class Foo {  
    public void publicMethod() {  
        ...  
        privateMethod();  
        ...  
    }  
    private void privateMethod() {  
        ...  
    }  
}
```

アクセス修飾子の目安

- ▶ 「アクセス範囲はなるべく狭くする」のが基本
 - ▶ 他のクラスで使わないメソッド→private
 - ▶ 同一パッケージのみで使うメソッド→指定なし
 - ▶ サブクラスでのみ使うメソッド→protected
 - ▶ 各サブクラスに共通する処理を提供するメソッド、サブクラスでオーバーライドしてほしいメソッドなど
 - ▶ 様々なクラスから呼び出すメソッド→public
- ▶ クラスも同様

publicなクラスのファイル名

- ▼ ソースファイル内にpublicなクラスが記述されていた場合、ソースファイル名はクラス名と一致させなければならない

Hoge.java

```
public class Hoge {  
    // ...  
}
```


カプセル化

カプセル化とは

- ➡ 複雑さを隠蔽し利用を簡単にする
- ➡ 不正な操作を防ぐ
- ➡ 修正に対して影響を極小化する

setter/getterの作成

➡ フィールドは基本的に
privateにする

➡ setter/getterを用いて
アクセスする

➡ setter/getterは基本
的にpublic

```
public class EmployeeEC {  
    private int id;  
    private String name;  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public void introduce() {  
        System.out.println("社員番号=" + id);  
        System.out.println("氏名=" + name);  
    }  
}
```

setter/getterの利用

▼フィールドへの値の代入は、setterを利用する

```
public class EmployeeECMain {  
    public static void main(String[] args) {  
        EmployeeEC emp = new EmployeeEC();  
        emp.setId(10001);  
        emp.setName("内田太郎");  
        emp.introduce();  
    }  
}
```

実行結果

```
C:\Yjava>java EmployeeECMain  
社員番号=10001  
氏名=内田太郎
```

setter/getterをつけるメリット

- ▶ 不正なアクセスを制限できる
 - ▶ 正の値しか入ってはいけないフィールドに、負の値が入るのを防ぐなど
- ▶ 実装を柔軟に変えられる
 - ▶ フィールドのデータ型が変わったとしても、setter/getterの実装だけを変えればよいので、他のクラスへの影響は少ない
- ▶ 安全性が高くなる
 - ▶ 引数やフィールドが参照型変数の場合、インスタンスをコピーしてから渡すなど

カプセル化していない場合

```
class Foo {  
    int id;  
}
```

```
class Foo {  
    String id;  
}
```

```
Foo f = new Foo();  
f.id = 100;
```

変更が
必要

```
Foo f = new Foo();  
f.id = "100";
```

カプセル化している場合

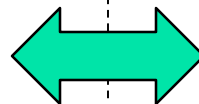
```
class Foo {  
    private int id;  
    public void setId(int id) {  
        this.id = id;  
    }  
    public int getId() {  
        return id;  
    }  
}
```

```
Foo f = new Foo();  
f.setId(100);
```

```
class Foo {  
    private String id;  
    public void setId(int id) {  
        this.id = String.valueOf(id);  
    }  
    public int getId() {  
        return Integer.parseInt(id);  
    }  
}
```

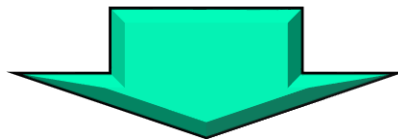
```
Foo f = new Foo();  
f.setId(100);
```

変更
不要



setter/getterに関する議論

- ▶ 「フィールドはpublicで良いのでは？」という意見も存在する
 - ▶ setter/getter不要論
- ▶ フレームワークによっては、setter/getterがないと動作しないものもある
 - ▶ 逆に、publicフィールドで動作するフレームワークもある



プロジェクトの規約を確認し、それに従う

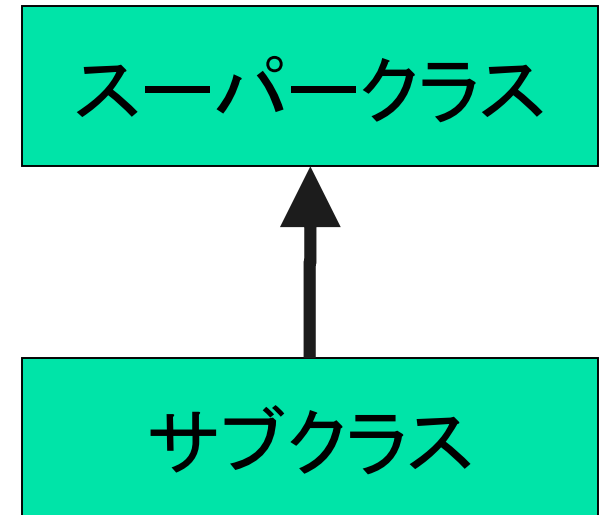
第5章

継承

継承の基本

継承とは

- ▶ 親クラス(スーパークラス)のメソッドやフィールドを引き継いだ子クラス(サブクラス)を作成すること



- ▶ extendsキーワード
 - ▶ extendsの後のクラスを継承していることを表す

サンプルプログラム

```
public class Human {  
    public String name;  
    public void introduce() {  
        System.out.println("氏名=" + name);  
    }  
}
```

```
public class EmployeeEX extends Human {  
    public int id;  
    public void work() {  
        System.out.println("仕事をします");  
    }  
}
```

Humanクラス



EmployeeEXクラス

サンプルプログラム

```
public class EmployeeEXMain {  
    public static void main(String[] args) {  
        EmployeeEX emp = new EmployeeEX();  
        emp.name = "内田太郎";  
        emp.id = 10001;  
        emp.introduce();  
        emp.work();  
    }  
}
```

スーパークラスに
定義されているので
利用可能

実行結果

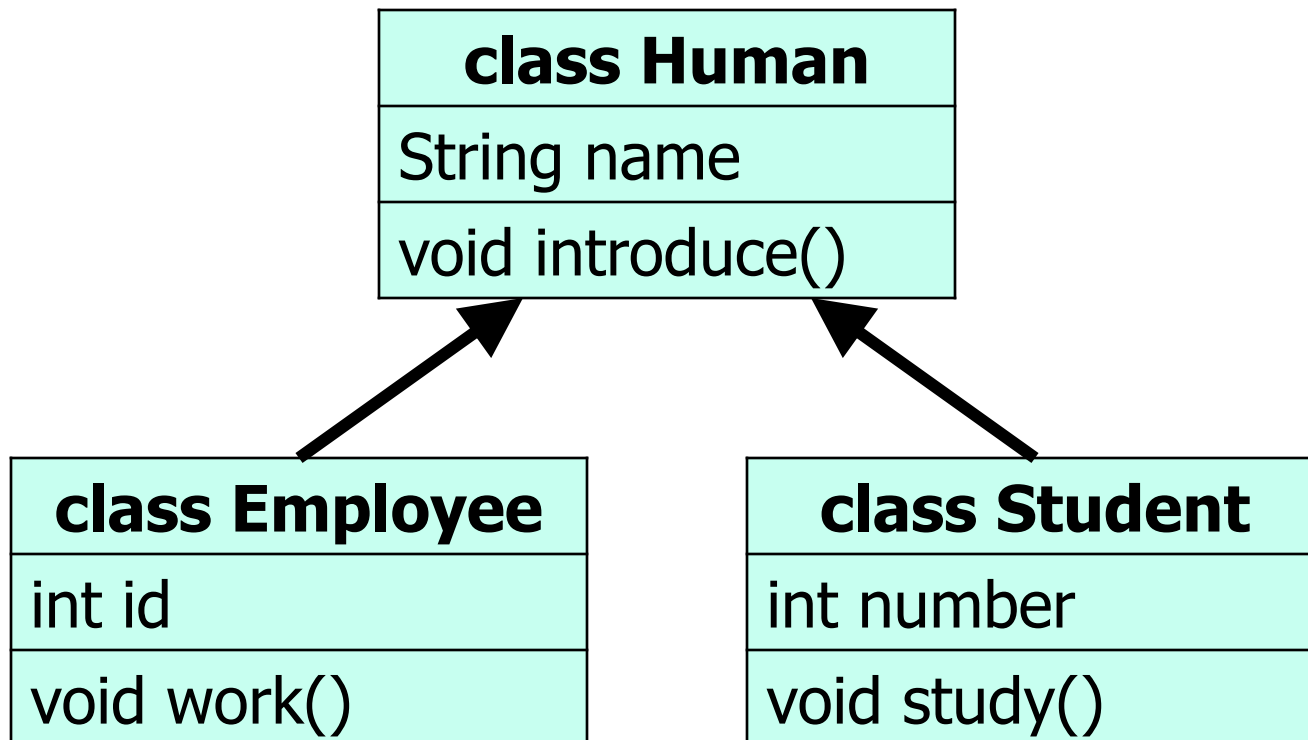
```
C:\¥java>java EmployeeEXMain  
氏名＝内田太郎  
仕事をします
```

継承のポイント

- ▶ スーパークラスのメンバは、サブクラスでもそのまま利用できる
- ▶ サブクラスでメンバを追加できる

継承のメリット

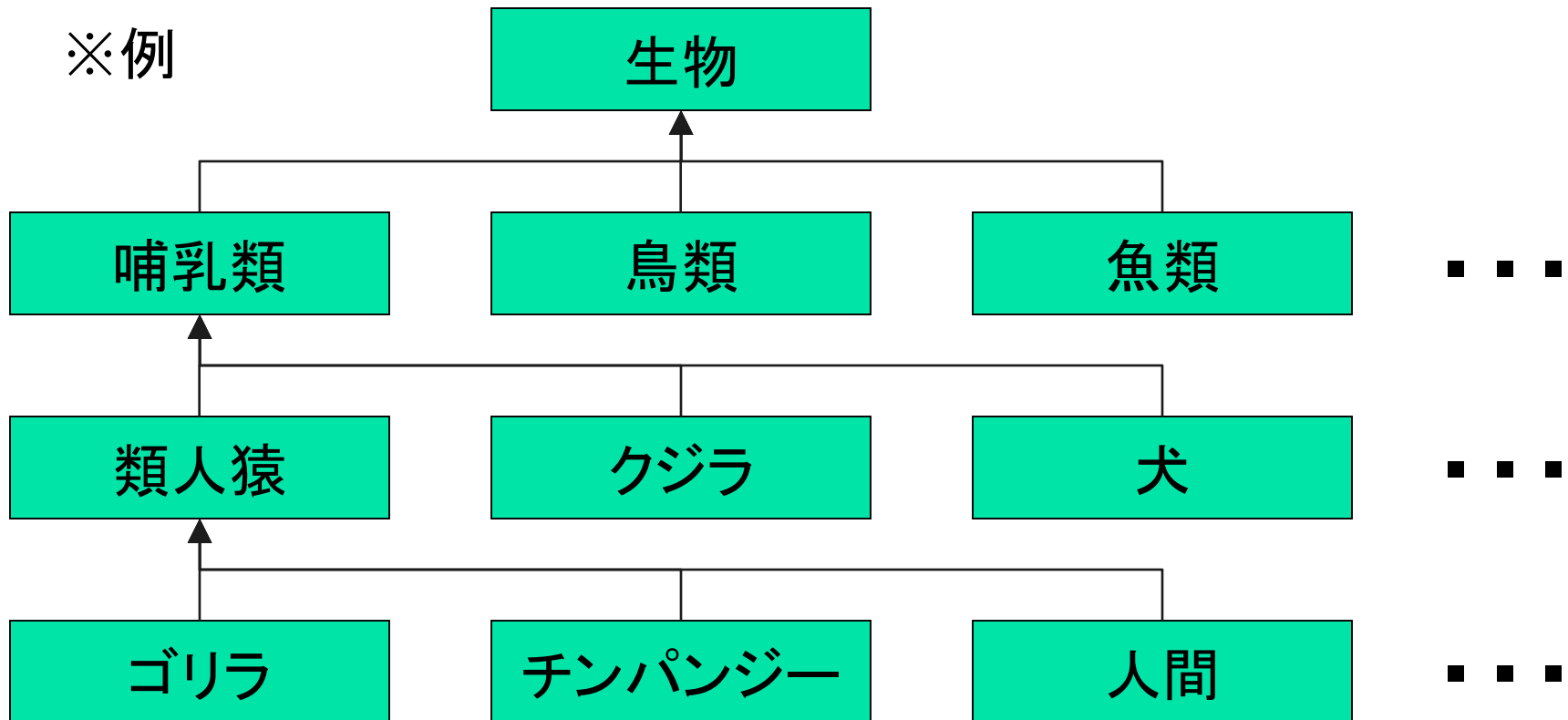
- ▶ 複数のクラスで共通するメンバを、スーパークラスにまとめることができる



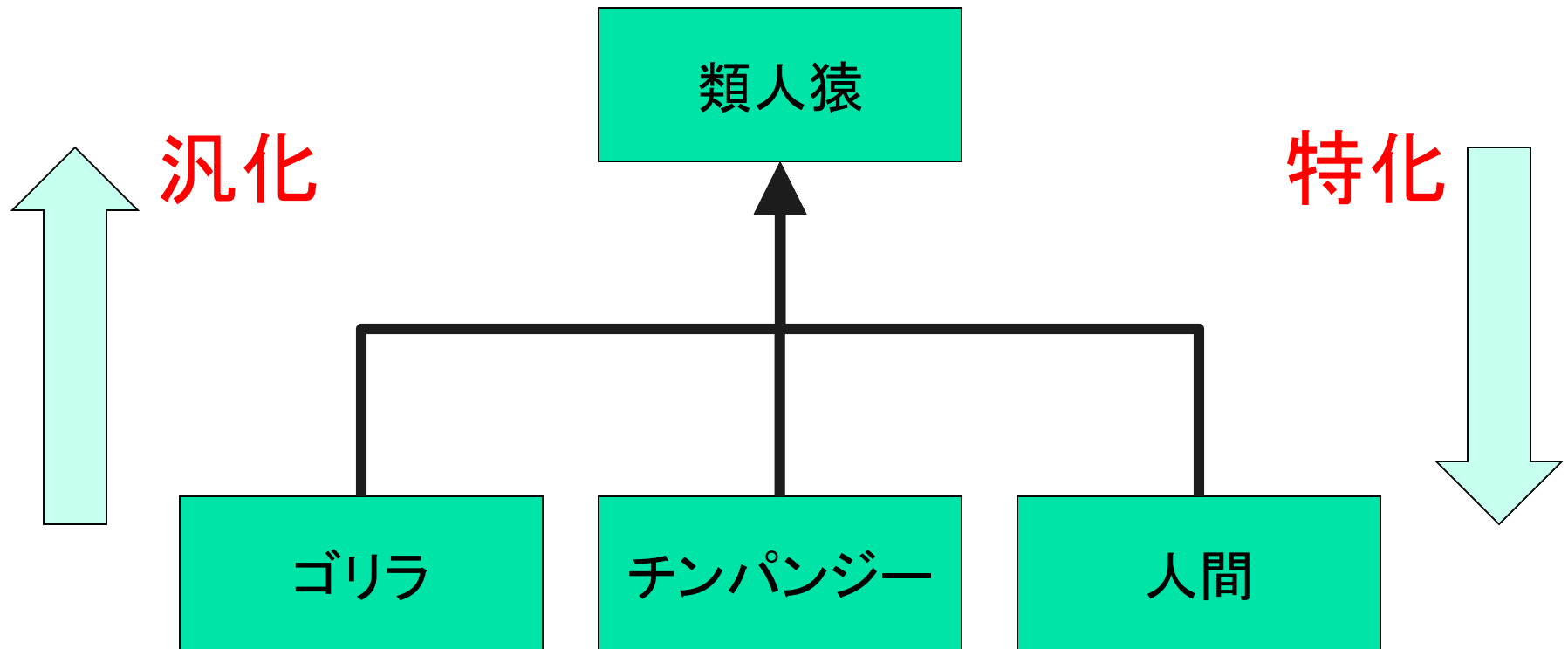
継承のイメージ

- ▶ サブクラスになればなるほど、**細分化・具体化**されるイメージ

※例



特化と汎化



is-aとhas-a

◀ ネコ is a 哺乳類

- ▶ 「ネコクラスは哺乳類クラスのサブクラス」
→ 継承関係

```
class 哺乳類{}  
class ネコ extends 哺乳類 {}
```

◀ 車 has a タイヤ

- ▶ 「タイヤクラスは、車クラスのフィールド」

```
class タイヤ {}  
class 車 {  
    タイヤ t;  
    ...  
}
```

問題

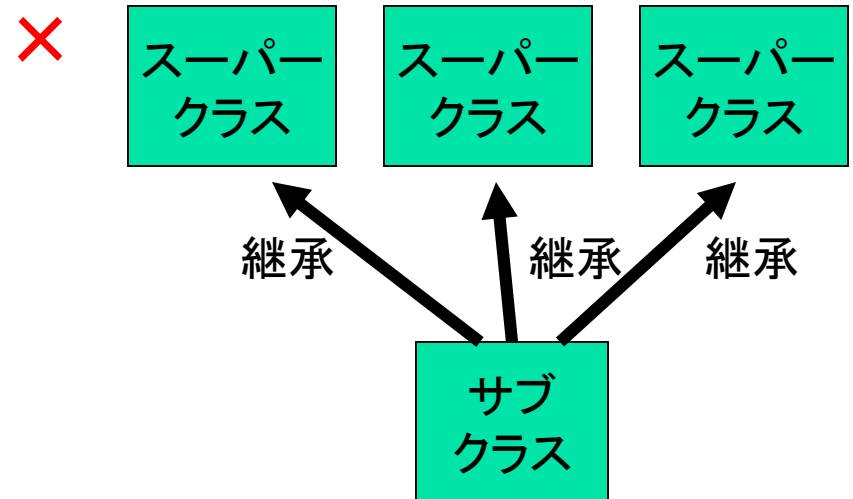
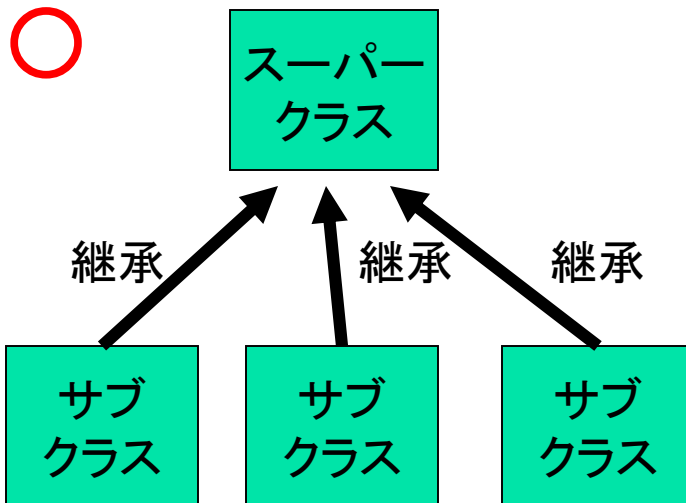
◀ 次のクラスと継承関係になるものは？

スーパークラス	サブクラス
(例) 魚	マグロ、サケ、イワシ...
(例) 犬	チワワ、ブルドッグ、ダックスフント...
(例) 車	トラック、バス、タクシー...
時計	
筆記用具	
コンピュータ	

単一継承

Javaでは、1つのクラスしか継承できない(単一継承)

C++などでは、複数のクラスを継承することが可能(多重継承)



継承の禁止

- ▶ クラスに**final**修飾子が付いていた場合、そのクラスのサブクラスを作ることができない

```
public final class Hoge {  
    ...  
}
```

```
public class Fuga extends Hoge { // ×コンパイルエラー  
    ...  
}
```

メソッドのオーバーライド

オーバーライドとは

- ➡ スーパークラスのメソッドと同じシグネチャのメソッドを**上書き定義**すること

```
public class EmployeeOR extends Human {  
    public int id;  
    public void work() {  
        System.out.println("仕事をします");  
    }  
    @Override  
    public void introduce() {  
        super.introduce();  
        System.out.println("社員番号=" + id);  
    }  
}
```

サンプルプログラム

```
public class EmployeeORMain {  
    public static void main(String[] args) {  
        EmployeeOR emp = new EmployeeOR();  
        emp.name = "内田太郎";  
        emp.id = 10001;  
        emp.introduce();  
        emp.work();  
    }  
}
```

実行結果

オーバーライドした
introduce()が
実行される

```
C:\¥java>java EmployeeORMain  
氏名＝内田太郎  
社員番号＝10001  
仕事をします
```


@Overrideアノテーション

- ▶ オーバーライドするメソッドにつけ、オーバーライドしていることを示す
 - ▶ Java SE 5.0で追加された機能
- ▶ 正確にオーバーライド出来ていなければ、コンパイルエラーとなる

superキーワード

- ▶ 親クラスへの参照を表す
- ▶ スーパークラスのメンバにアクセス
 - ▶ super.<フィールド名>、super.<メソッド名>
- ▶ 自クラスのメンバにアクセス
 - ▶ this.<フィールド名>、this.<メソッド名>

オーバーライドの用途

- ▶ 後述の「ポリモーフィズム」のための使うことが多い

オーバーライドの禁止

- メソッドに**final**修飾子が付いていた場合、そのメソッドをオーバーライドできない

```
public class Hoge {  
    public final void method() { ... }  
}
```

```
public class Fuga extends Hoge {  
    @Override  
    public void method() { ... } // ×コンパイルエラー  
}
```

【参考】オーバーライドのその他のルール

➡ 戻り値

- ➡ 元のメソッドと同じか、そのサブタイプでなければならない

➡ アクセス修飾子

- ➡ 元のメソッドと同じか、より広くなければならない

➡ 例外

- ➡ 例外をスローする場合、元のメソッドと同じか、そのサブクラス例外でなければならない
- ➡ 例外をスローしなくてもよい
- ➡ RuntimeExceptionのサブクラスは、自由にスローしてよい

継承とコンストラクタ

継承とコンストラクタ

- ▶ スーパークラスのコンストラクタは継承されない
 - ▶ サブクラスでコンストラクタを定義しなければならない
- ▶ スーパークラスのコンストラクタを使用するには`super()`キーワードを使用する

```
public class HumanCS {  
    public String name;  
    public HumanCS(String name) {  
        this.name = name;  
    }  
    public void introduce() {  
        System.out.println("氏名=" + name);  
    }  
}
```

```
public class EmployeeEXCS extends HumanCS {  
    public int id;  
    public EmployeeEXCS(String name, int id) {  
        super(name);  
        this.id = id;  
    }  
    public void work() {  
        System.out.println("仕事をします");  
    }  
}
```

サンプルプログラム

```
public class EmployeeEXCSMain {  
    public static void main(String[] args) {  
        EmployeeEXCS emp =  
            new EmployeeEXCS("内田太郎", 10001);  
        emp.introduce();  
        emp.work();  
    }  
}
```

実行結果

```
C:\¥java>java EmployeeEXCSMain  
氏名＝内田太郎  
仕事をします
```


super()の暗黙的呼び出し

- ▶ `super()` (スーパークラスのデフォルトコンストラクタ) は、記述していなくても暗黙的に呼び出される
- ▶ 明示的に記述する場合、サブクラスのコンストラクタ内の先頭に書かなければならない

その他のトピックス

【復習】アクセス修飾子

広

アクセス可能範囲

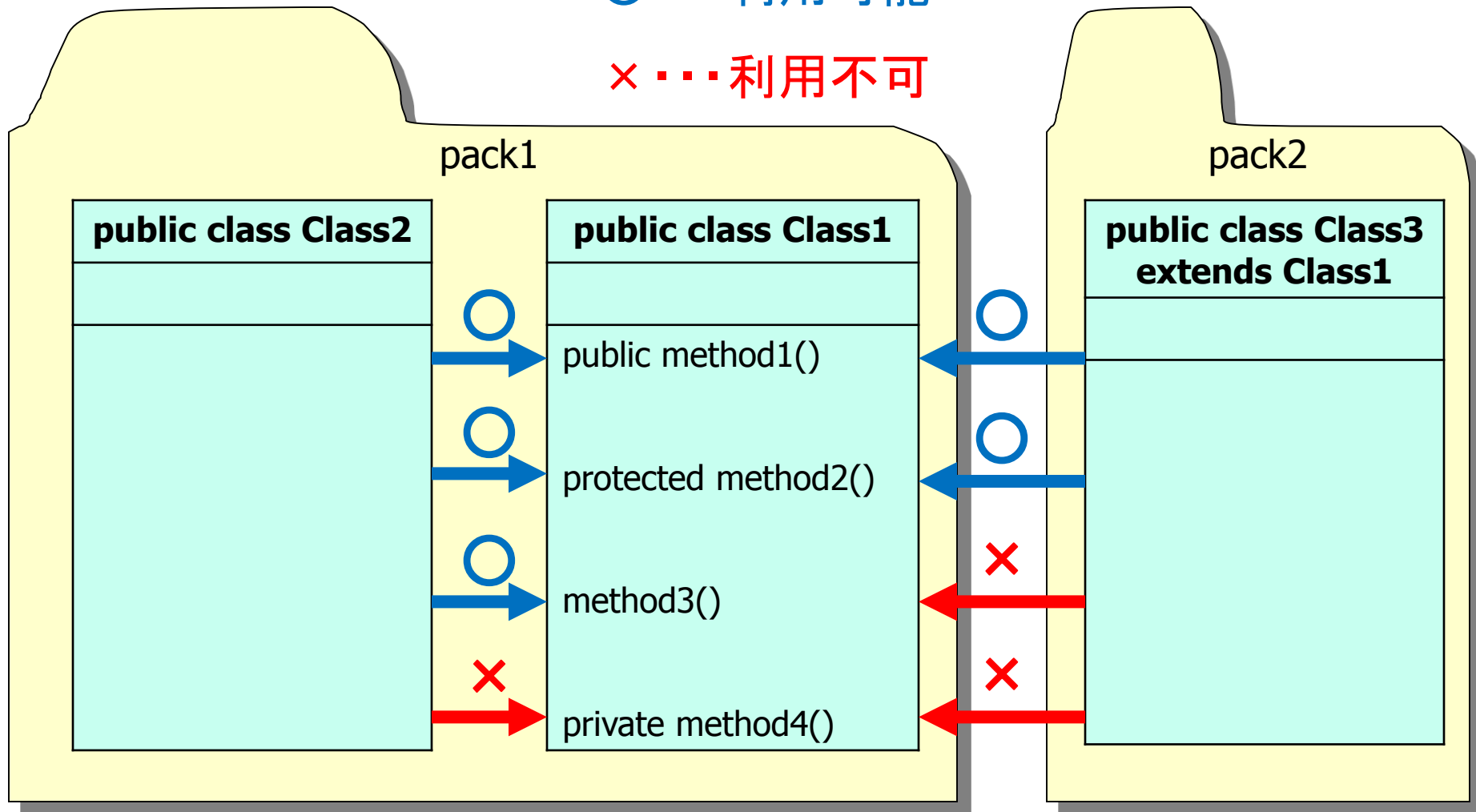
狭

アクセス修飾子	説明
public	パッケージ等に関わらず、すべてのクラスからアクセスできる
protected	同一パッケージ内のクラス、またはサブクラスからアクセスできる
指定なし(デフォルト、パッケージプライベート)	同一パッケージ内のクラスからのみアクセスできる
private	同じクラス内からのみアクセスできる

継承とアクセス修飾子

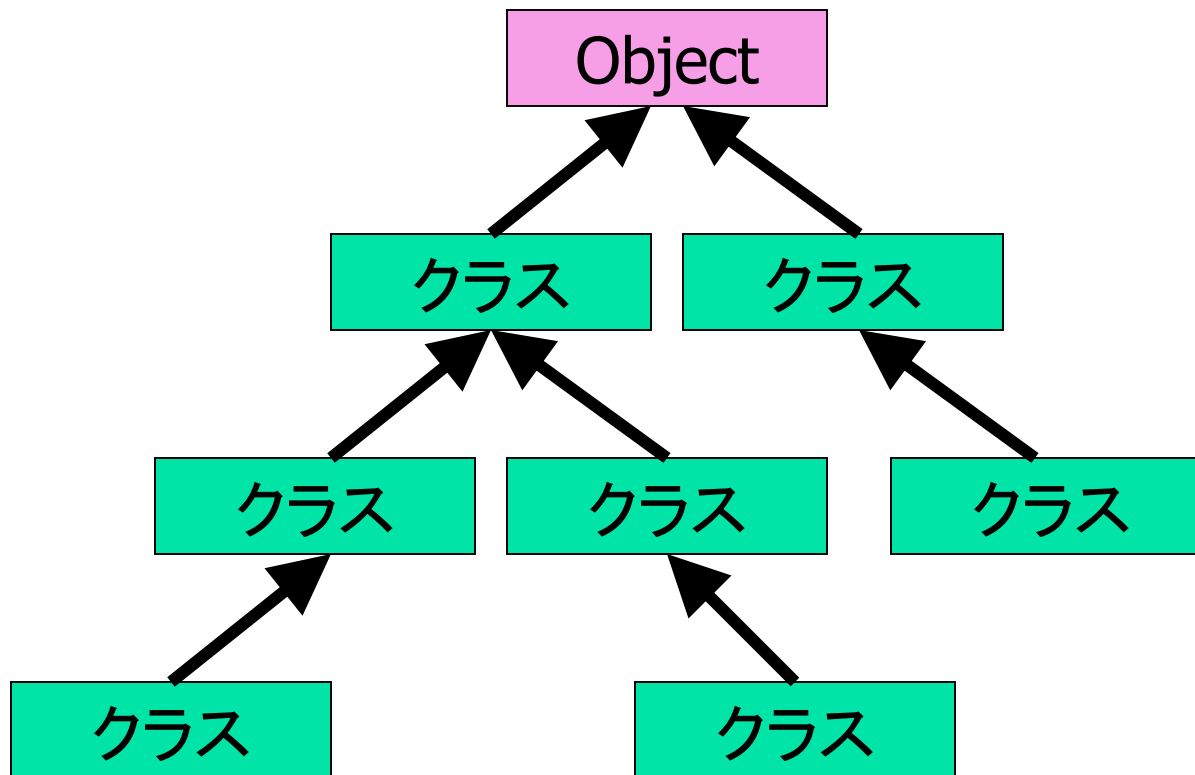
○・・・利用可能

×・・・利用不可



Objectクラス

すべてのクラスのスーパークラス



Objectクラス(つづき)

- ▶ extends節を記述していない場合、Objectクラスのサブクラスと見なされる

```
class Foo {  
    ...  
}
```

=

```
class Foo extends Object {  
    ...  
}
```

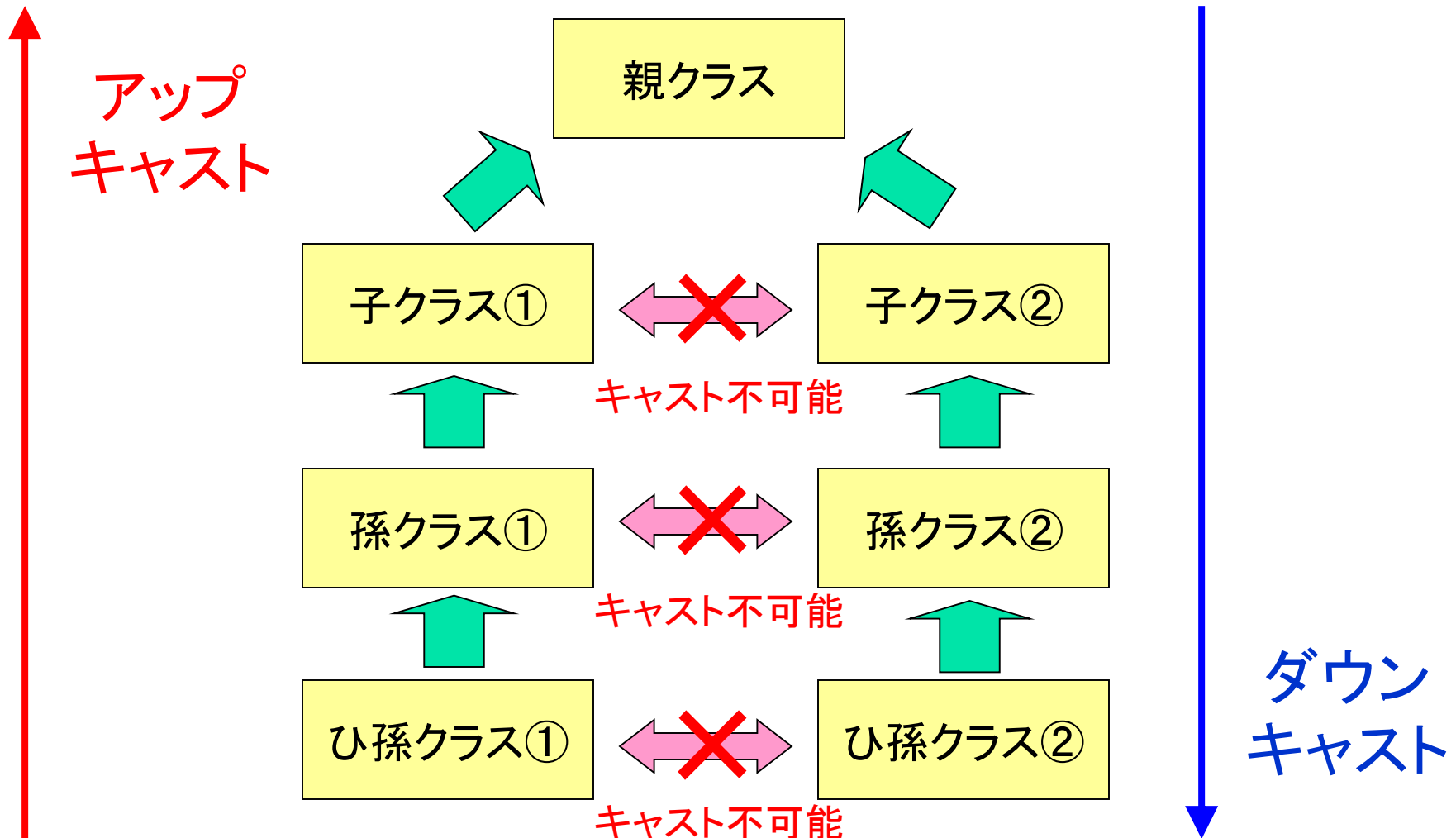
コンパイラが自動で
追加する

第6章

ポリモーフィズム

クラスの型変換

クラスのキャスト順位



型変換

- ▶ アップキャストは、暗黙的に行われる
- ▶ ダウンキャストは、明示的に行わないとコンパイルエラーとなる
- ▶ 継承関係が無い場合、キャストは不可能

```
class Parent {}
```

```
class Child extends Parent {}
```

```
Parent p = new Child(); // ○
```

```
Child c = (Child) p; // ○
```

```
Child c = p; // × (コンパイルエラー)
```

```
String s = p; // × (コンパイルエラー)
```

ClassCastException

- 不正なキャストを行った際、実行時に発生する例外

```
Parent p = new Parent();  
Object obj = p;  
String s = (String) obj;
```

実行結果

```
Exception in thread "main" java.lang.ClassCastException:  
    Parent cannot be cast to java.lang.String  
at Foo.main(Foo.java:7)
```

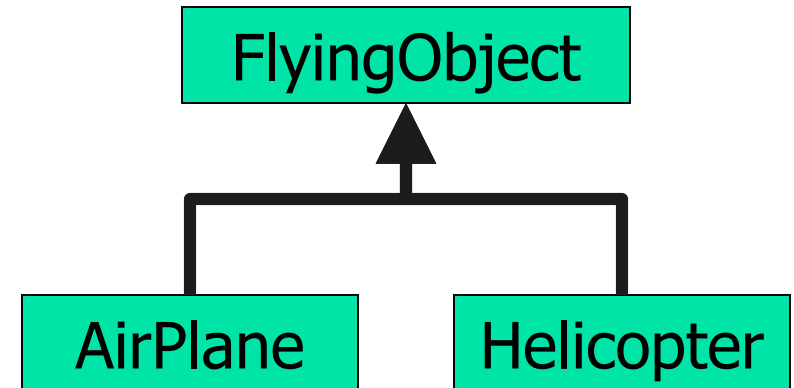
ポリモーフィズムの基本

サンプルプログラム

```
public class FlyingObject {  
    public void fly() {  
        System.out.println("飛びます");  
    }  
}
```

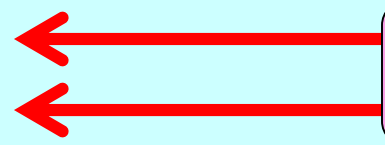
```
public class AirPlane extends FlyingObject {  
    @Override  
    public void fly() {  
        System.out.println("エンジンと翼で飛びます");  
    }  
}
```

```
public class Helicopter extends FlyingObject {  
    @Override  
    public void fly() {  
        System.out.println("プロペラで飛びます");  
    }  
}
```



サンプルプログラム

```
public class FlyingObjectMain {  
    public static void main(String[] args) {  
        FlyingObject fo1 = new AirPlane();  
        FlyingObject fo2 = new Helicopter();  
        fo1.fly();  
        fo2.fly();  
    }  
}
```



オーバーライドしたメソッドが
実行される

実行結果

```
C:\¥java>java FlyingObjectMain  
エンジンと翼で飛びます  
プロペラで飛びます
```

呼び出されるメソッド

▶ 呼び出されるメソッドは「**インスタンスの型**」に従う

▶ 「**変数の型**」ではない

```
class Parent {  
    void foo() { System.out.println("Parent#foo()"); }  
}  
  
class Child extends Parent {  
    @Override  
    void foo() { System.out.println("Child#foo()"); }  
}
```

```
Parent p1 = new Parent();  
p1.foo(); // Parent#foo()  
  
Parent p2 = new Child();  
p2.foo(); // Child#foo()
```

呼び出し可能なメソッド

呼び出せるメソッドは「**変数の型**」に従う

```
class Parent {  
    void foo() { System.out.println("Parent#foo()"); }  
}  
  
class Child extends Parent {  
    @Override  
    void foo() { System.out.println("Child#foo()"); }  
    // メソッド追加  
    void bar() { System.out.println("Child#bar()"); }  
}
```

```
Child c = new Child();  
c.foo(); // Child#foo()  
c.bar(); // Child#bar()  
  
Parent p = new Child();  
p.foo(); // Child#foo()  
p.bar(); // コンパイルエラー
```


ポリモーフィズムのメリット

▶ 変更に強い

- ▶ 変更が生じた場合、プログラム中の変更箇所が少なく済む
- ▶ サブクラスの実装を変えればよく、それを使う側の変更は少ない

▶ 使う側が楽

- ▶ サブクラスの実装を気にしなくてもプログラムが作成できる

抽象クラス

抽象クラスとは

- ▶ 抽象クラスの**インスタンス**は**生成できない**
 - ▶ 抽象クラス型の**変数は定義できる**
- ▶ クラスや抽象メソッドに**abstract**キーワードを指定
 - ▶ 抽象メソッド: 実装を持たないメソッド

// 抽象クラス

```
abstract class AbHoge {  
    // 抽象メソッド  
    abstract void foo();  
    void bar() {  
        System.out.println("BAR");  
    }  
}
```

// コンパイルエラー

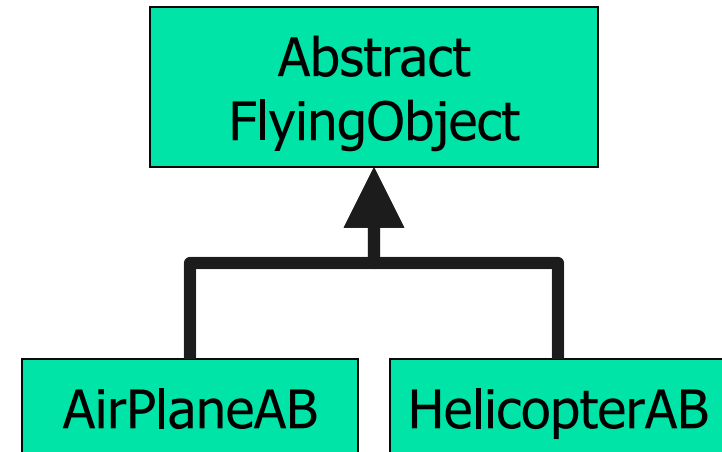
```
AbHoge hoge = new AbHoge();
```

サンプルプログラム

```
public abstract class AbstractFlyingObject {  
    public abstract void fly();  
}
```

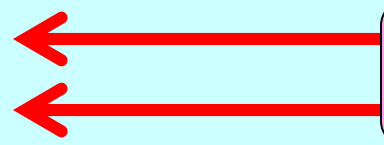
```
public class AirPlaneAB extends AbstractFlyingObject {  
    @Override  
    public void fly() {  
        System.out.println("エンジンと翼で飛びます");  
    }  
}
```

```
public class HelicopterAB extends AbstractFlyingObject {  
    @Override  
    public void fly() {  
        System.out.println("プロペラで飛びます");  
    }  
}
```



サンプルプログラム

```
public class AbstractFlyingObjectMain {  
    public static void main(String[] args) {  
        AbstractFlyingObject fo1 = new AirPlaneAB();  
        AbstractFlyingObject fo2 = new HelicopterAB();  
        fo1.fly();  
        fo2.fly();  
    }  
}
```



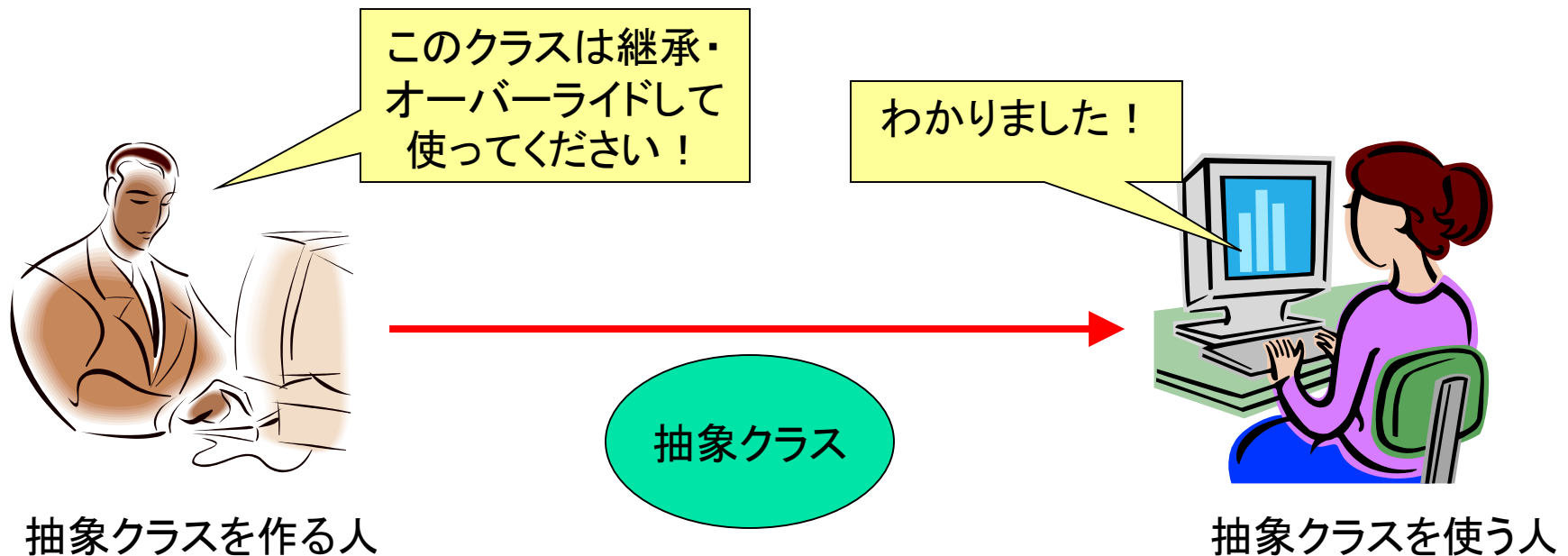
オーバーライドしたメソッドが
実行される

実行結果

```
C:\¥java>java AbstractFlyingObjectMain  
エンジンと翼で飛びます  
プロペラで飛びます
```

抽象クラスのメリット

- ▶ インスタンスが生成不可なので、「このクラスは継承・オーバーライドして使ってください」という明確なメッセージになる
- ▶ あくまでもクラスなので、共通機能をまとめることができる



抽象クラスに関する注意

- ▶ 抽象メソッドをprivateやfinalには出来ない
 - ▶ サブクラスでオーバーライド出来なくなるから
- ▶ 抽象クラスを継承したのに抽象メソッドを実装しなかった場合、コンパイルエラーになる
 - ▶ 解決策
 - ▶ メソッドを実装する
 - ▶ そのクラスも抽象クラスにする

インターフェイス

インターフェイスとは

- ▶ メソッドのシグネチャと戻り値のみを定義したもの
- ▶ メソッドはすべて抽象メソッド (**public abstract**)
- ▶ インスタンスは生成できない
 - ▶ インターフェイス型の変数は定義できる
- ▶ インターフェイスは**クラスではない**

```
// インターフェイス  
interface IFoo {  
    // 抽象メソッド  
    public void bar();  
}
```

```
// コンパイルエラー  
IFoo foo = new IFoo();
```

インターフェイスの使い方

- ▶ **implements** キーワードを用いて、インターフェイスを**実装**したクラスを作る

```
interface IFoo {  
    public void bar();  
}
```

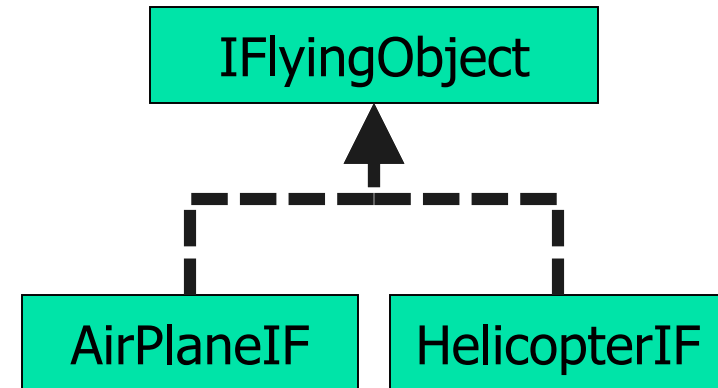
```
class Foo implements IFoo {  
    // 抽象メソッドを実装  
    @Override  
    public void bar() {  
        System.out.println("BAR");  
    }  
}
```

サンプルプログラム

```
public interface IFlyingObject {  
    public void fly();  
}
```

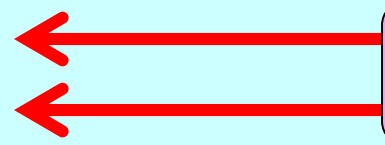
```
public class AirPlaneIF implements IFlyingObject {  
    @Override  
    public void fly() {  
        System.out.println("エンジンと翼で飛びます");  
    }  
}
```

```
public class HelicopterIF implements IFlyingObject {  
    @Override  
    public void fly() {  
        System.out.println("プロペラで飛びます");  
    }  
}
```



サンプルプログラム

```
public class IFlyingObjectMain {  
    public static void main(String[] args) {  
        IFlyingObject fo1 = new AirPlaneIF();  
        IFlyingObject fo2 = new HelicopterIF();  
        fo1.fly();  
        fo2.fly();  
    }  
}
```



オーバーライドしたメソッドが
実行される

実行結果

```
C:\¥java>java IFlyingObjectMain  
エンジンと翼で飛びます  
プロペラで飛びます
```

定数の定義

- ➡ フィールドはすべて定数
(= **public static final**)になる
- ➡ メソッドの戻り値として使うことが多い

```
interface IFoo {  
    int HOGE = 10;  
    int FUGA = 20;  
    // HOGEまたはFUGAを返す  
    public int method();  
}
```

複数のインターフェイスの実装

▶ インターフェイスは複数実装できる

```
class <クラス名> implements <インターフェイス1>, <インターフェイス2>, ... {  
    ...  
}
```

▶ インターフェイスの実装とクラスの継承は同時にできる

```
class <クラス名> extends <スーパークラス名> implements <インターフェイス1> {  
    ...  
}
```

インターフェイスの継承

▶ インターフェイスは継承できる

```
interface <インターフェイス名> extends <スーパーインターフェイス名> {  
    ...  
}
```

▶ インターフェイスは多重継承できる

```
interface <インターフェイス名> extends <スーパーインターフェイス1>, <スーパーインターフェイス2>, ... {  
    ...  
}
```

抽象クラスと比較した際の インターフェイスのメリット・デメリット

▶ メリット

▶ 実装を持たないので、より柔軟に実装を変える事が出来る。

▶ ただし、**外部から見た使い方** (メソッドの戻り値・シグネチャなど)を変えない範囲に限る

▶ デメリット

▶ 実装を持たないので、抽象クラスのように共通機能をまとめることが出来ない

第7章 パッケージ



パッケージ

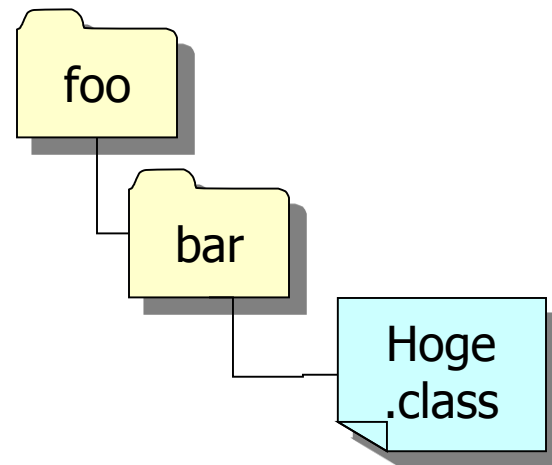
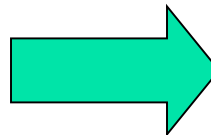
パッケージとは

- ▶ クラスファイルをフォルダ分けしたもの
- ▶ ソースファイルの先頭に「パッケージ宣言」を記述する
 - ▶ `package <パッケージ名>;`

Hoge.java

```
package foo.bar;  
class Hoge {  
    // ...  
}
```

コンパイル



クラスの完全修飾名

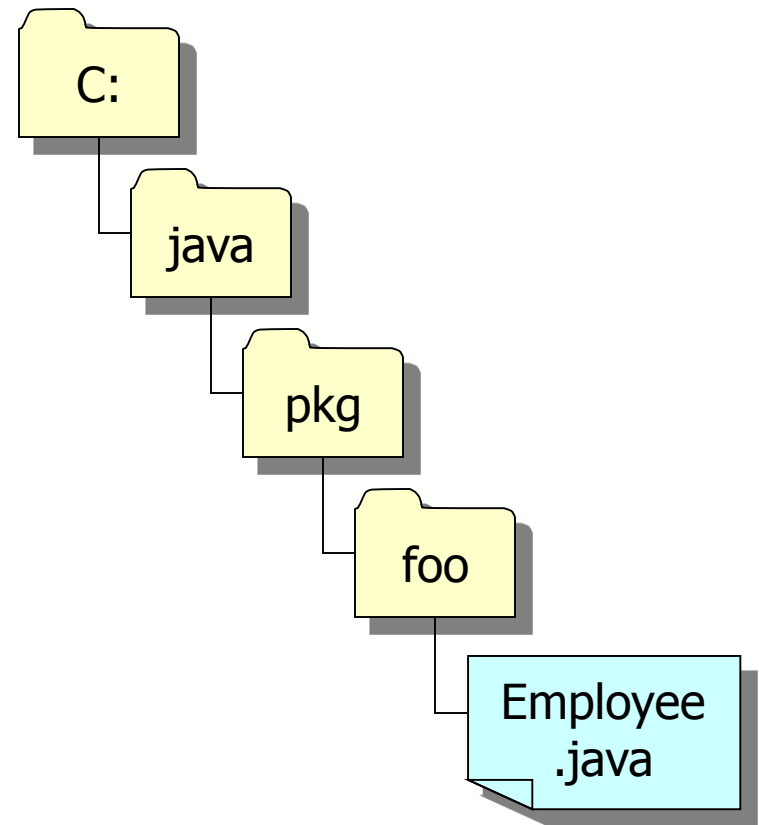
- ▶ 完全修飾名 = パッケージ名.クラス名
 - ▶ 例えば、foo.barパッケージのHogeクラスの完全修飾名はfoo.bar.Hoge

【手順1】パッケージつきクラスの作成

➡ C:¥java¥pkgフォルダの中に、以下のようなクラスを作成

➡ **パッケージ宣言**が必要

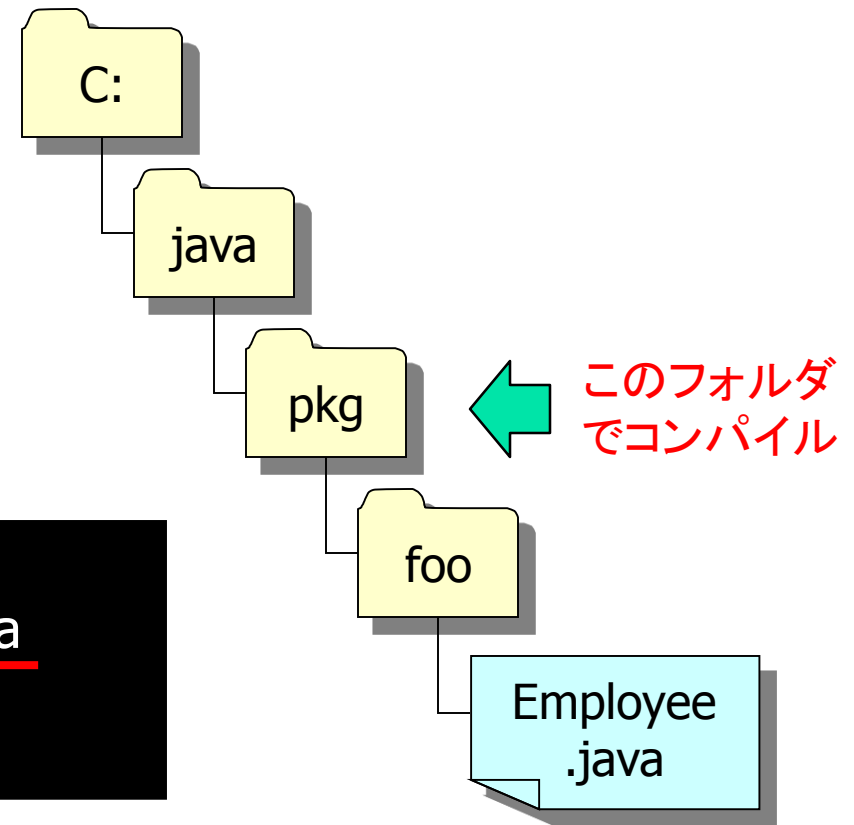
```
package foo; // パッケージ宣言
public class Employee {
    public void work() {
        System.out.println("働きます");
    }
}
```



【手順2】パッケージつきクラスのコンパイル

- ① パッケージのすぐ上のフォルダに移動
- ② javacの後に、ソースコードのパスを入力

```
C:¥>cd C:¥java¥pkg  
C:¥java¥pkg>javac .¥foo¥Employee.java  
               ソースコードのパス
```

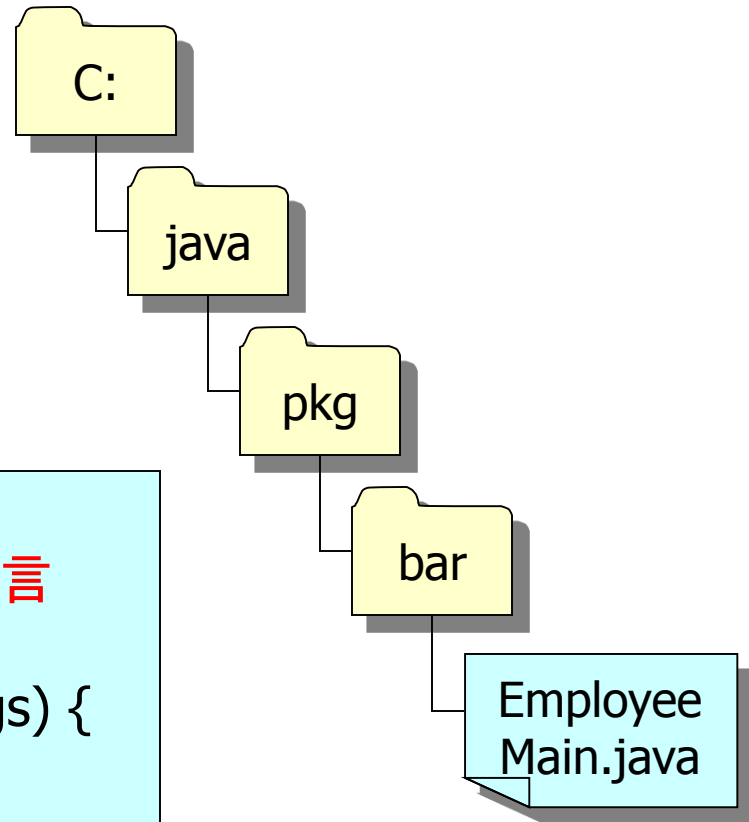


【手順3】パッケージつきクラスの利用

➡ C:¥java¥pkgフォルダの中に、以下のようなクラスを作成

➡ **インポート宣言**が必要

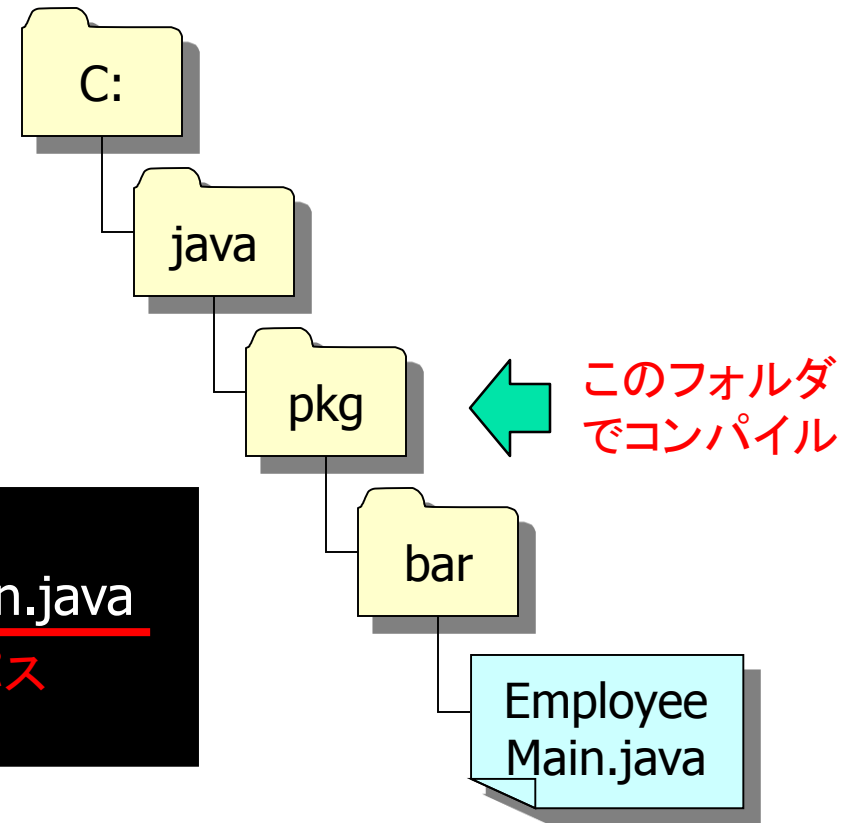
```
package bar;  
import foo.Employee; // インポート宣言  
public class EmployeeMain {  
    public static void main(String[] args) {  
        Employee e = new Employee();  
        e.work();  
    }  
}
```



【手順4】コンパイル

- ① パッケージのすぐ上のフォルダに移動
- ② javacの後に、ソースコードのパスを入力

```
C:¥>cd C:¥java¥pkg  
C:¥java¥pkg>javac ¥bar¥EmployeeMain.java  
ソースコードのパス
```

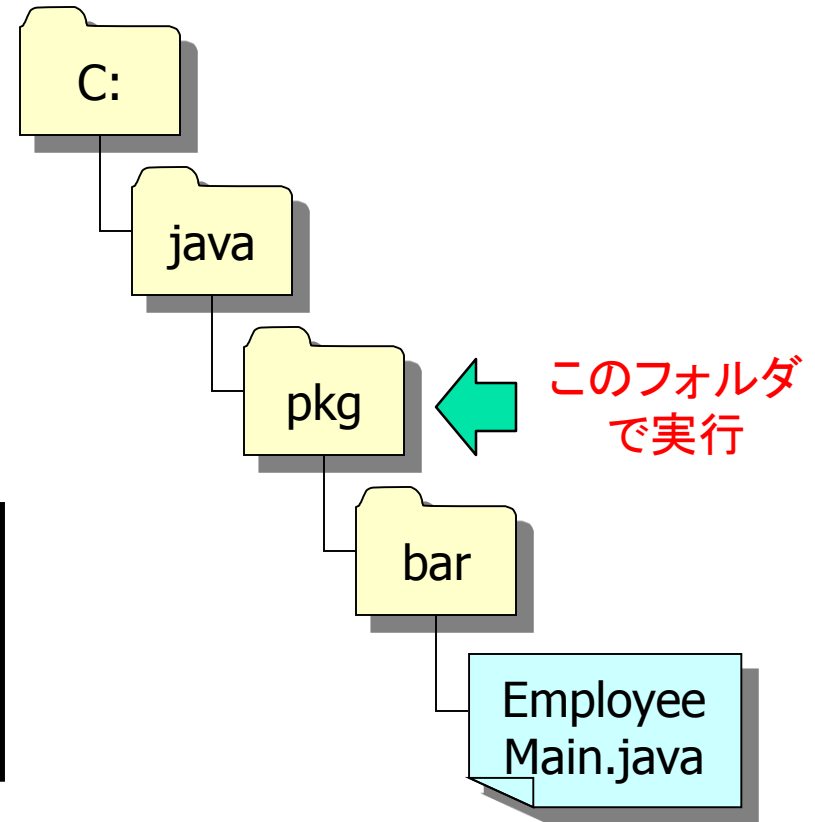


【手順5】実行

- ① パッケージのすぐ上のフォルダに移動
- ② javaの後に、**完全限定クラス名**を入力

```
C:¥>cd C:¥java¥pkg  
C:¥java¥pkg>java bar.EmployeeMain  
働きます
```

完全修飾名

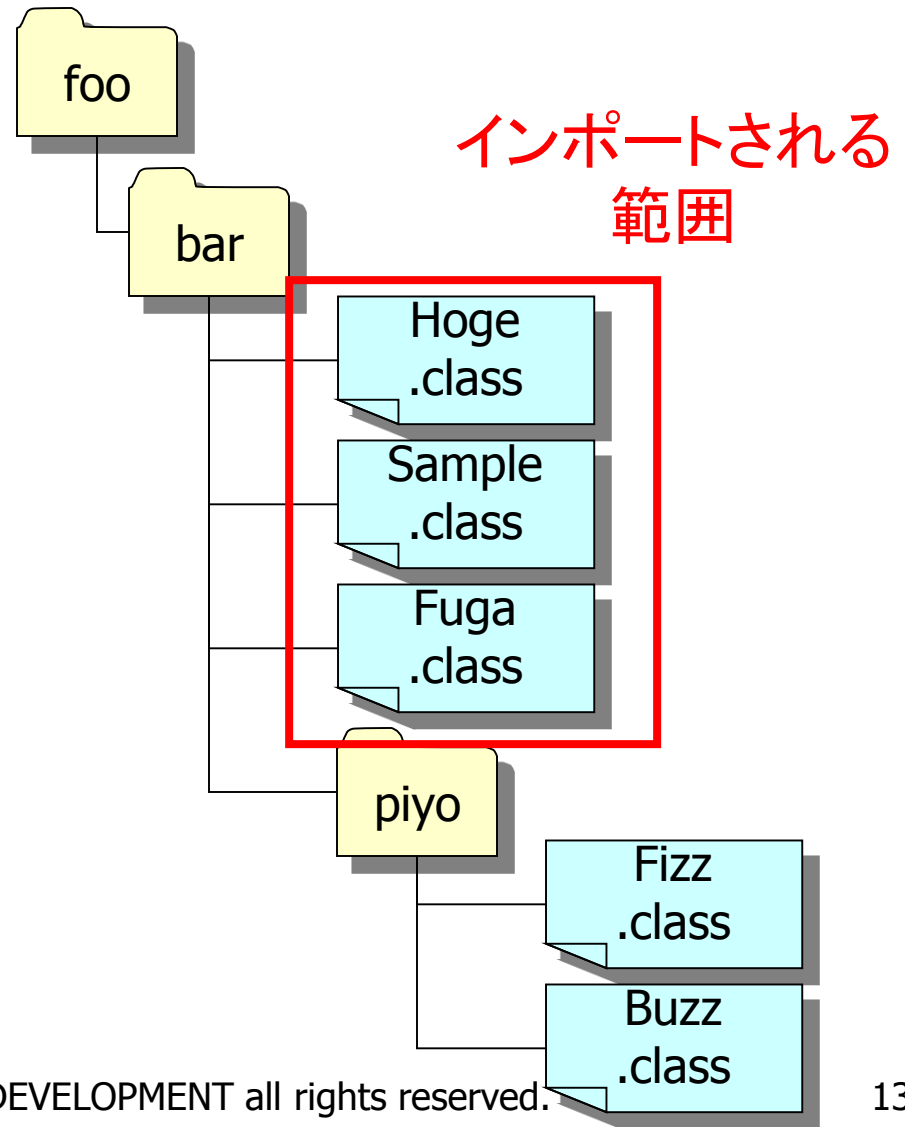


ポイントまとめ

- ▶ パッケージを作成するには、パッケージ宣言が必要
- ▶ 別パッケージのクラスを利用するには、インポート宣言が必要
- ▶ コンパイルは、パッケージのすぐ上のフォルダで、ソースコードのパスを指定
- ▶ 実行は、パッケージのすぐ上のフォルダで、クラスの完全修飾名を指定

インポートのワイルドカード指定

- ▶ `import foo.bar.*;`
とすれば、foo.barパッケージのクラスがすべてインポートされる。
- ▶ どのクラスを使用しているか分かりづらくなるため、プロジェクトによってはこの書き方を非推奨にしている場合もある。



ソースファイル内での記述順

▶ パッケージ宣言→インポート宣言→クラス定義の順

▶ 順番が違うとコンパイルエラー

```
// ①パッケージ宣言  
package foo.bar;  
  
// ②インポート宣言  
import foo.Hello;  
  
// ③クラス定義  
class Hoge {  
    // ...  
}
```

【復習】アクセス修飾子

広

アクセス可能範囲

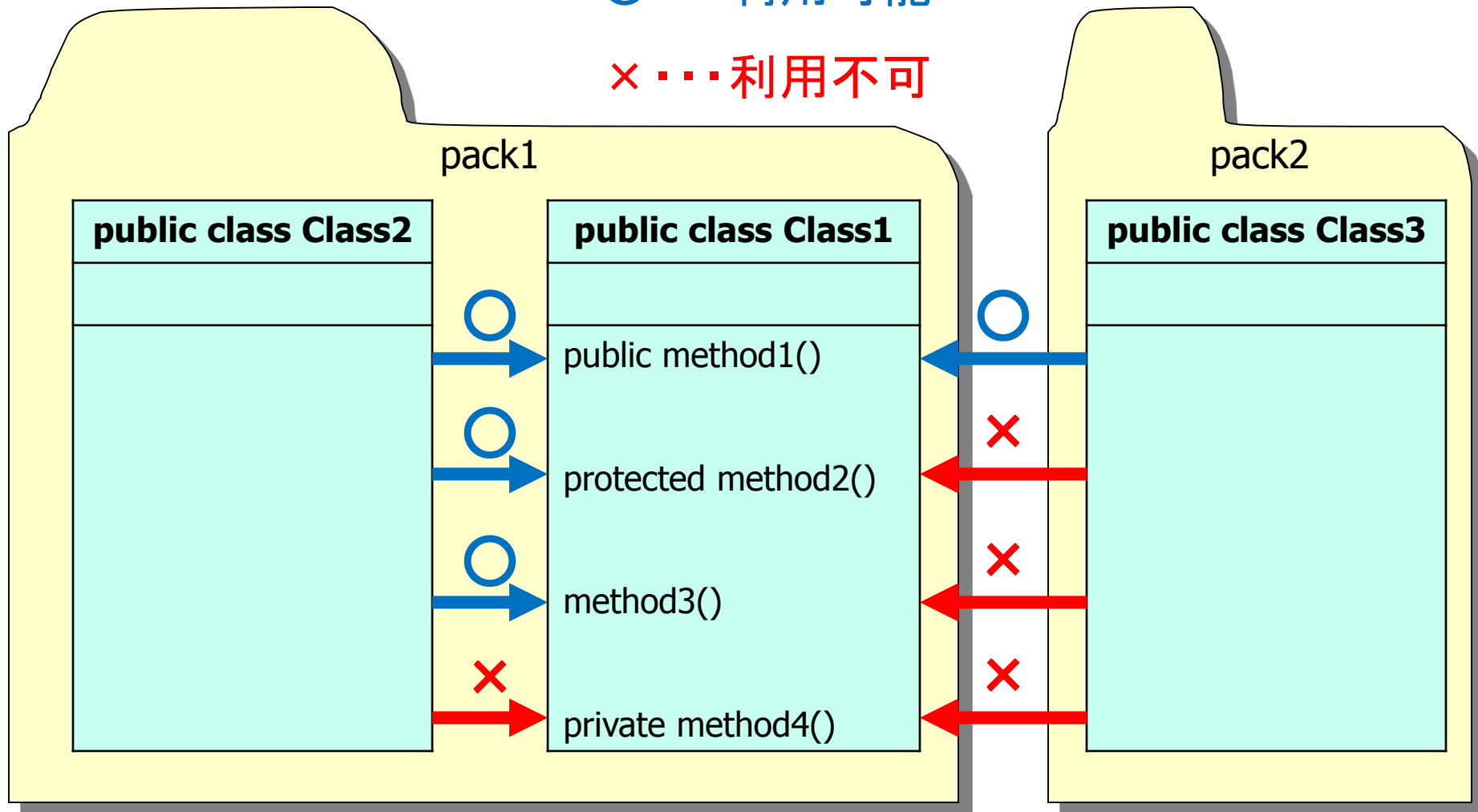
狭

アクセス修飾子	説明
public	パッケージ等に関わらず、すべてのクラスからアクセスできる
protected	同一パッケージ内のクラス、またはサブクラスからアクセスできる
指定なし(デフォルト、パッケージプライベート)	同一パッケージ内のクラスからのみアクセスできる
private	同じクラス内からのみアクセスできる

パッケージとアクセス修飾子(メンバ)

○・・・利用可能

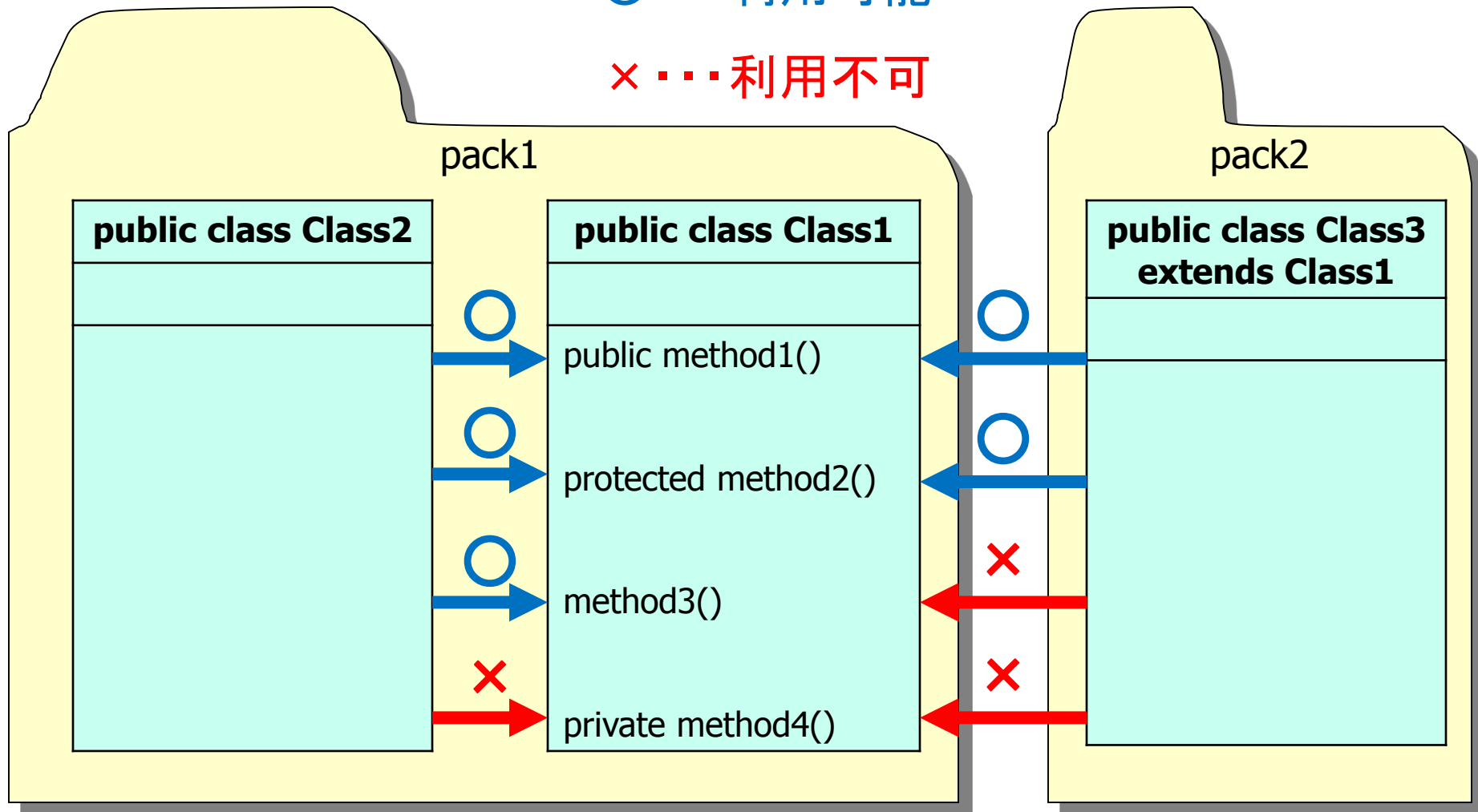
×・・・利用不可



パッケージとアクセス修飾子(メンバ・継承関係あり)

○・・・利用可能

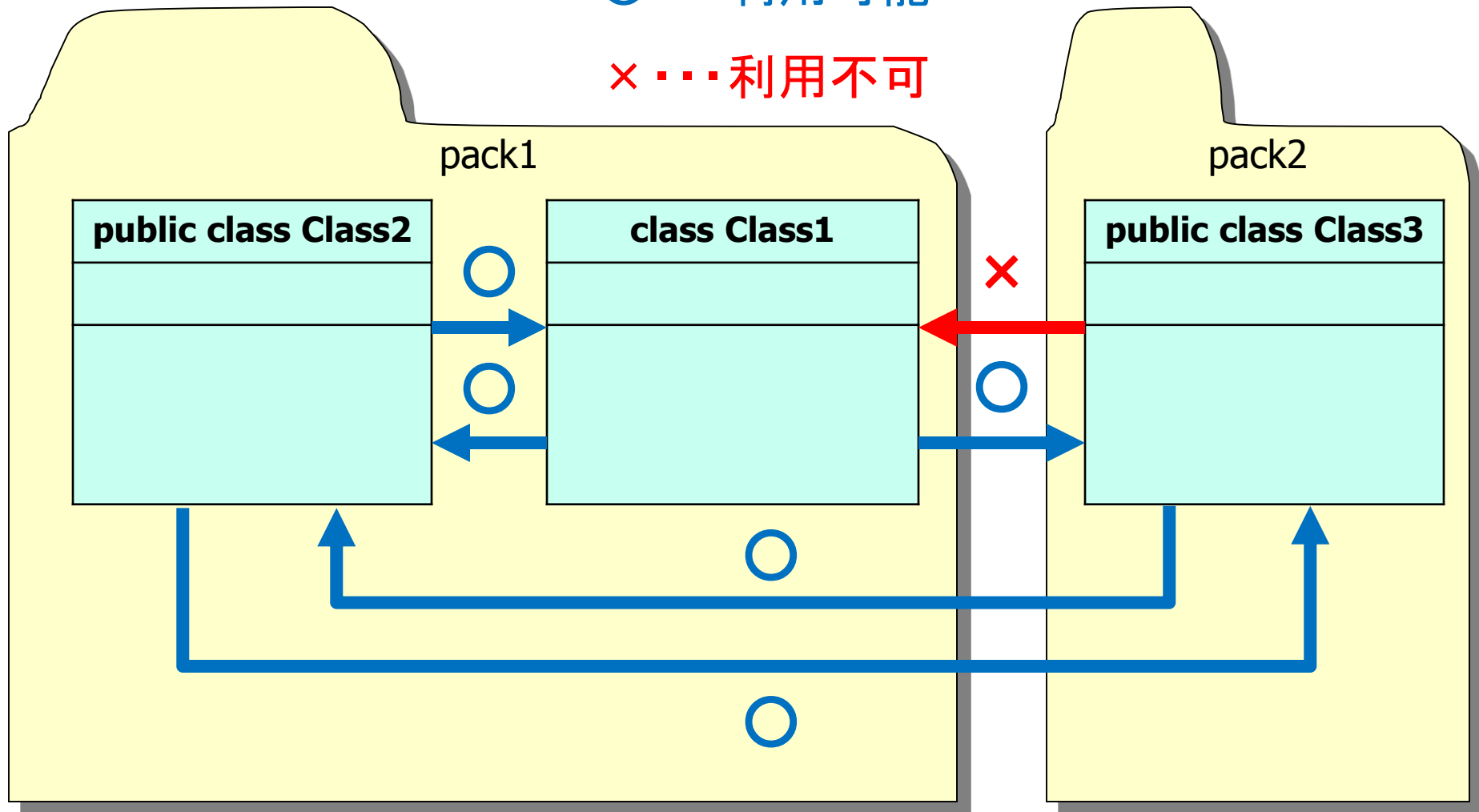
×・・・利用不可



パッケージとアクセス修飾子(クラス)

○・・・利用可能

×・・・利用不可

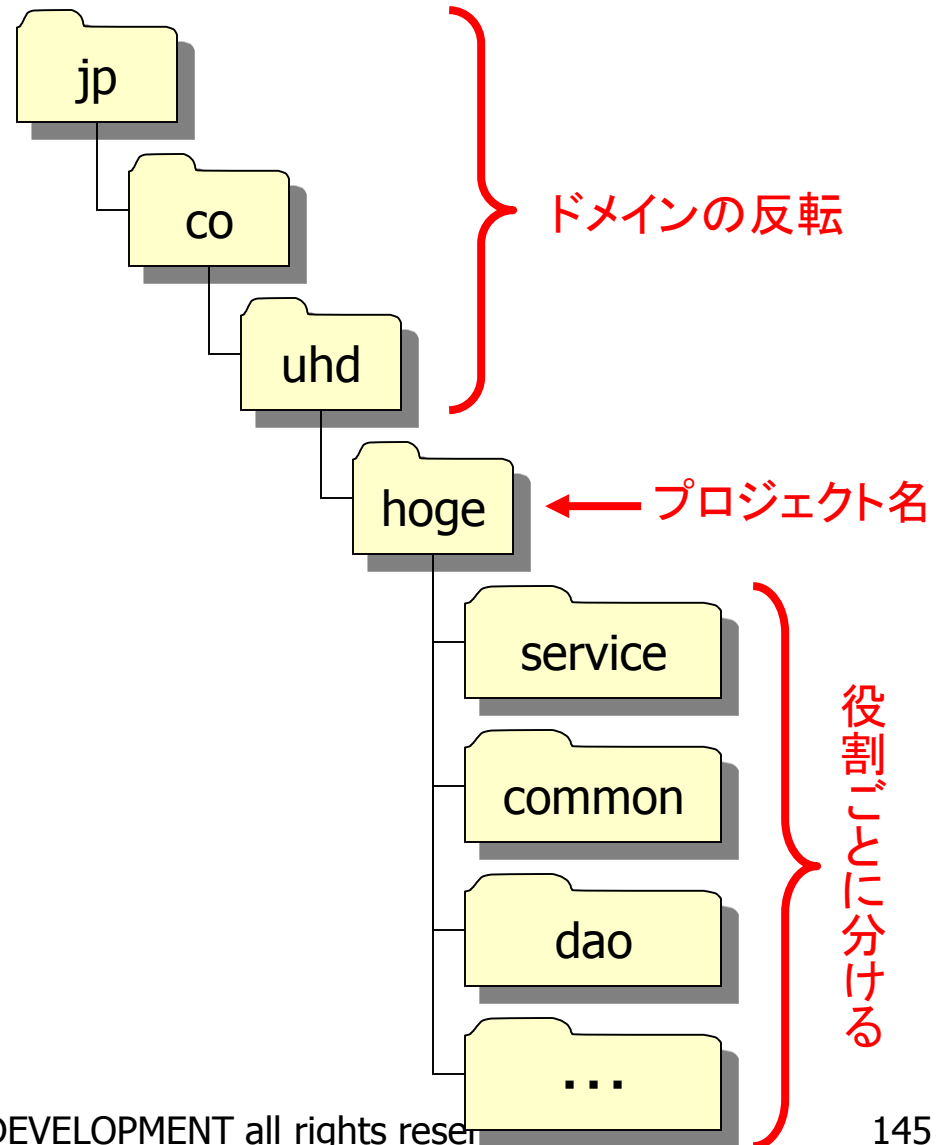


パッケージ名のつけ方

- 自社のドメイン名を反転させてつけることが多い

- UHD社(ドメイン: uhd.co.jp)のプロジェクト「HOGE」の例

- クラス名をユニーク(世界で唯一)にするため





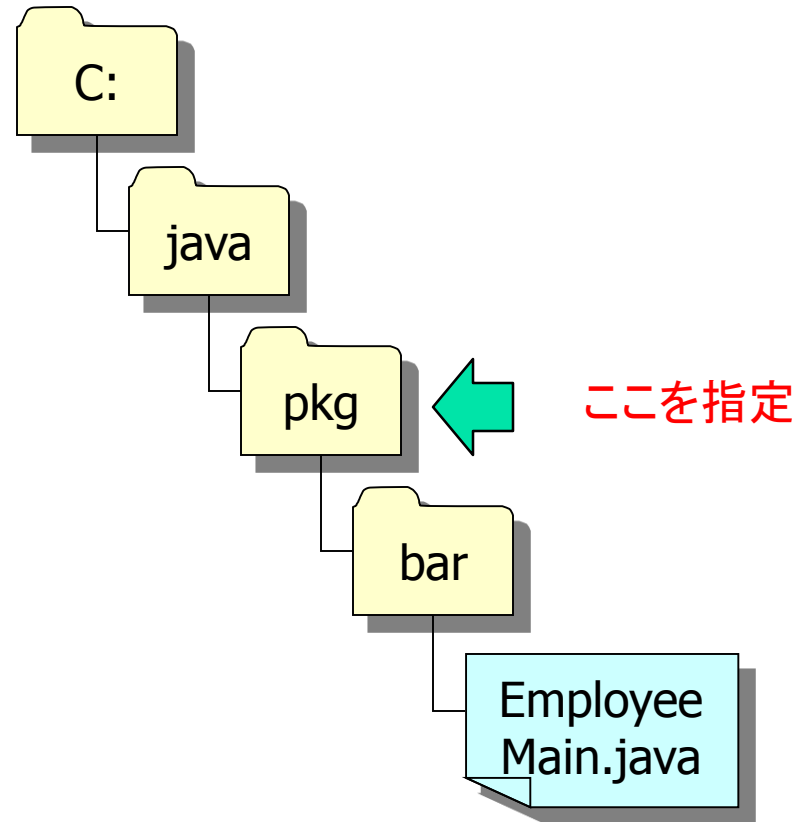
クラスパス

クラスパスとは

- ▶ JVMがクラスファイルを探すパス
 - ▶ デフォルトはカレントディレクトリ
- ▶ コマンドでのクラスパス指定方法
 - ▶ 環境変数で指定
 - ▶ `set CLASSPATH = <クラスパス>`
 - ▶ オプションで指定
 - ▶ `java -cp <クラスパス> <実行クラス>`

パッケージとクラスパスの関係

- ▶ クラスパスは、パッケージのすぐ上のフォルダを指定



クラスパスを指定して実行

```
C:¥java¥pkg>cd C:¥  
C:¥>java -cp C:¥java¥pkg bar.EmployeeMain  
働きます
```

-cpの後に、クラスファイル
があるフォルダを指定

※クラスパス指定をしなかった場合

```
C:¥java¥pkg>cd C:¥  
C:¥>java bar.EmployeeMain  
Exception in thread "main"  
java.lang.NoClassDefFoundError: bar/EmployeeMain  
(以下略)
```



実行すべきクラ
スが見つからず
例外発生

クラスパスが複数の場合

▶ セミコロン「;」で区切って指定する

```
C:¥java¥pkg>cd C:¥  
C:¥>java -cp C:¥java¥pkg;C:¥hoge piyo.FugaMain
```

セミコロンで区切る

JARファイル

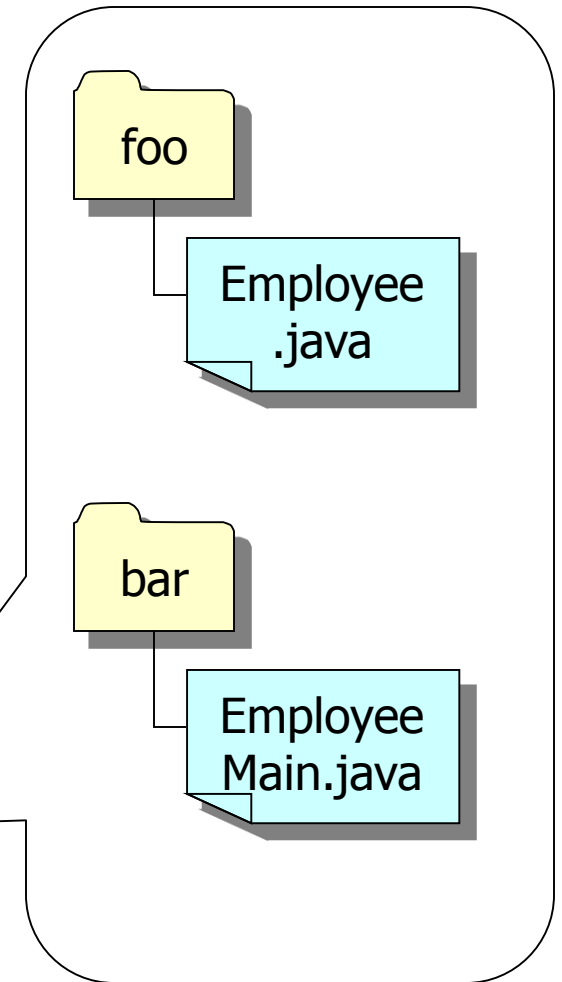
JAR (Java ARchive) ファイルとは

- ▶ クラスファイルなどを、パッケージごとZIP形式で圧縮したもの
 - ▶ 自作クラスなどが配布しやすい
 - ▶ オープンソースのツールなどは、ほとんどJAR形式でWebで配布している

JARファイルの作成

▶ 「jar -cvf <jarファイル名>
<パス>」で作成実行

```
C:\java\pkg>jar -cvf emp.jar .  
マニフェストが追加されました。  
bar/ を追加中です。（以下略）
```



JARファイルの利用

▼ JARファイルをクラスパスに指定する

```
C:¥java¥pkg>java -cp emp.jar bar.EmployeeMain  
働きます
```

第8章

例外処理

例外の基本

例外とは

- ▶ プログラムのミスやその他外部環境などが原因で、実行時に発生する異常
 - ▶ 配列の長さを超えてアクセスしようとした
 - ▶ 0で割り算を行おうとした
 - ▶ 文字列型を整数型に変換する際、許されない文字列が入力された
 - ▶ 読み込むべきファイルが見つからなかった
 - ▶ ネットワーク上で異常が起きた
 - ▶ DBに異常が発生し、接続できなかった
 - ...など

サンプルプログラム

```
public class ExceptionSample1 {  
    public static void main(String[] args) {  
        int[] a = new int[3];  
        System.out.println("値を代入します");  
        a[3] = 30; // 例外発生  
        System.out.println("処理を終了します");  
    }  
}
```

実行結果

```
C:\¥java>java ExceptionSample1  
値を代入します  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3  
at ExceptionSample1.main(ExceptionSample1.java:5)
```

スタックトレース

どこでどんな例外が発生したかを表示する

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3  
at ExceptionSample1.main(ExceptionSample1.java:5)
```

例外の発生場所

at クラス名.メソッド名(ファイル名:行番号)

例外クラスの名前

例外の種類ごとに違う
メッセージが出る

メソッド内での例外発生

```
public class ExceptionSample2 {  
    public static void main(String[] args) {  
        System.out.println("mainを開始します");  
        method();  
        System.out.println("mainを終了します");  
    }  
    static void method() {  
        System.out.println("メソッドを開始します");  
        int[] a = new int[3];  
        a[3] = 30; // 例外発生  
        System.out.println("メソッドを終了します");  
    }  
}
```

実行結果

```
C:\¥java>java ExceptionSample2  
mainを開始します  
メソッドを開始します  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3  
at ExceptionSample2.method(ExceptionSample2.java:10)  
at ExceptionSample2.main(ExceptionSample2.java:4)
```


複数行のスタックトレース

- main以外のメソッド内で例外が発生した場合、スタックトレースが複数行出力される

```
at ExceptionSample2.method(ExceptionSample2.java:10)
```

最初に例外が
発生したメソッド

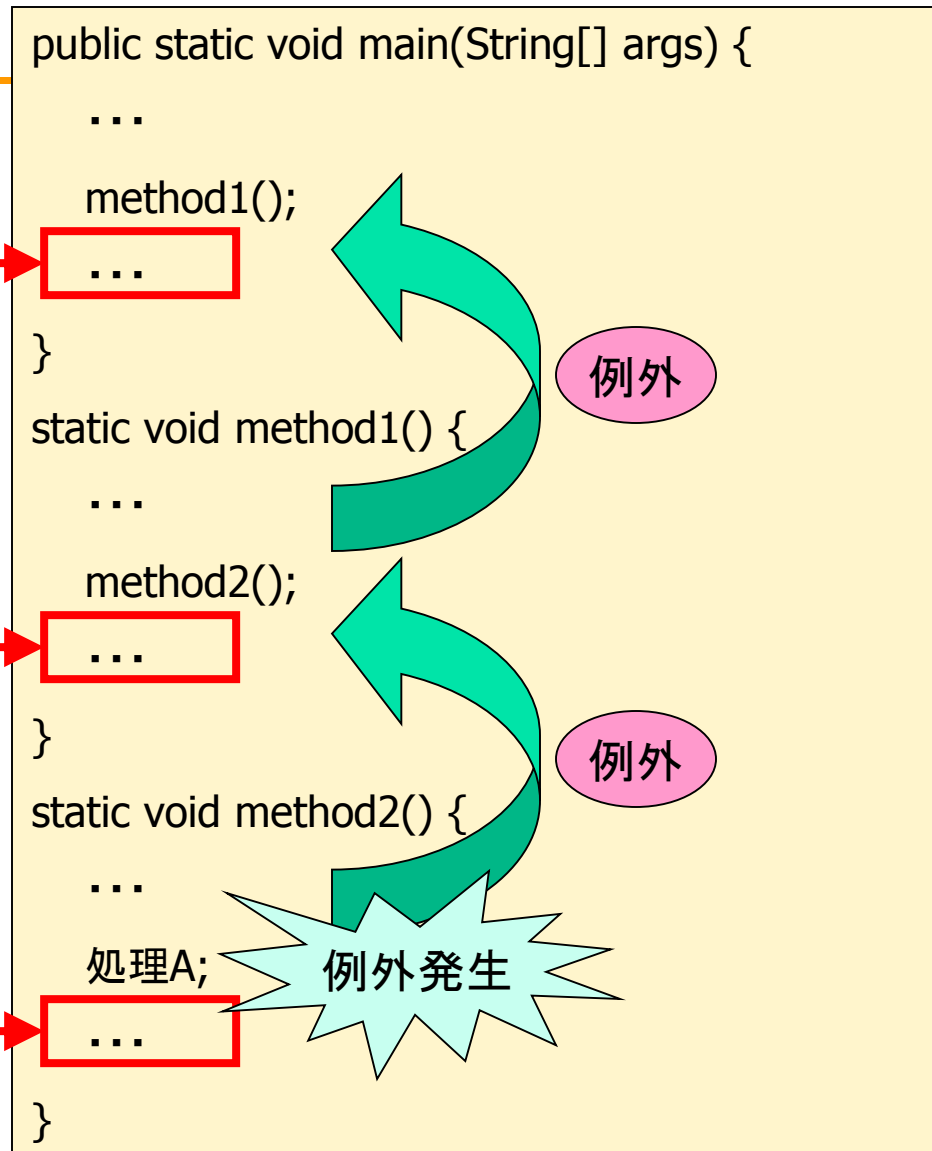
```
at ExceptionSample2.main(ExceptionSample2.java:4)
```

メソッドの呼び出し元

例外のスロー

- ▶ 例外が発生すると、メソッド内のそれ以降の処理を中断し、呼び出しもとのメソッドに処理を戻す
＝例外をスローする

処理が行われない



標準APIメソッドでの例外発生

```
public class ExceptionSample3 {  
    public static void main(String[] args) {  
        String str = "abc";  
        int num = Integer.parseInt(str); // 例外発生  
        System.out.println(num);  
    }  
}
```

実行結果

```
C:\¥java>java ExceptionSample3  
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"  
at java.lang.NumberFormatException.forInputString(Unknown Source)  
at java.lang.Integer.parseInt(Unknown Source)  
at java.lang.Integer.parseInt(Unknown Source)  
at ExceptionSample3.main(ExceptionSample3.java:4)
```

主な例外の種類

例外の種類	主な発生要因
ArrayIndexOutOfBoundsException	配列の長さを超えるインデックスを指定した
NumberFormatException	Integer#parseIntなどの引数に、数値でない文字列を指定した
ArithmeticException	0での割り算を行った
InputMismatchException	Scanner#nextIntによる入力において、数値でない文字列を指定した
NullPointerException	nullである変数のフィールド・メソッドを利用しようとした
ClassCastException	継承関係の無い型にキャストしようとした
IOException	入出力時に何らかの異常が発生した
FileNotFoundException	存在しないファイルを指定した
SQLException	データベースに関する異常が発生した

try-catch-finallyによる例外処理

try-catch文

▶ 例外が発生した場合の処理を記述する

```
try {  
    例外が発生する可能性がある処理  
} catch (<例外型> <変数名>) {  
    例外が発生した場合の処理  
}
```

サンプルプログラム

```
public class TryCatchSample1 {  
    public static void main(String[] args) {  
        System.out.println("mainを開始します");  
        try {  
            int[] a = new int[3];  
            System.out.println("値を代入します");  
            a[3] = 30; // 例外発生  
            System.out.println("処理を終了します");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("例外が発生しました");  
            e.printStackTrace();  
        }  
        System.out.println("mainを終了します");  
    }  
}
```

実行結果

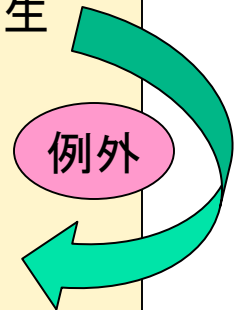
```
C:¥java>java TryCatchSample1  
mainを開始します  
値を代入します  
例外が発生しました  
(スタックトレース)  
mainを終了します
```

処理の流れ

try節の中で例外が発生した場合、

1. try節の中の以降の処理は中断する
2. 該当する型のcatch節の中の処理を実行する
 - それ以外の型のcatch節は実行されない
3. catch節より後の処理を実行する

```
処理1;  
try {  
    処理2;  
    処理3; // FooException発生  
    処理4;  
} catch (FooException fe) {  
    例外処理1;  
} catch (BarException be) {  
    例外処理2;  
}  
処理5;
```



例外クラスのメソッド

▶ public void printStackTrace()

▶ スタックトレースを表示する

finally節

- ▶ 例外が発生の有無にかかわらず、必ず行う処理を記述する

```
try {  
    例外が発生する可能性がある処理  
} catch (<例外型> <変数名>) {  
    例外が発生した場合の処理  
} finally {  
    必ず行う処理  
}
```

サンプルプログラム

```
public class TryCatchSample2 {  
    public static void main(String[] args) {  
        System.out.println("mainを開始します");  
        try {  
            int[] a = new int[3];  
            System.out.println("値を代入します");  
            a[3] = 30; // 例外発生  
            System.out.println("処理を終了します");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("例外が発生しました");  
            e.printStackTrace();  
        } finally {  
            System.out.println("必ず行う処理");  
        }  
        System.out.println("mainを終了します");  
    }  
}
```

実行結果

```
C:\¥java>java TryCatchSample2  
mainを開始します  
値を代入します  
例外が発生しました  
(スタックトレース)  
必ず行う処理  
mainを終了します
```

変数のスコープ

- try、catch、finallyなどの各ブロック内で宣言した変数のスコープは、そのブロック内
- ブロック外では使えない

```
int a = 10;           aのスコープ
...
try {
    int b = 20;
    ...               bのスコープ
} catch (<例外型> <変数名>) {
    int c = 30;
    ...               cのスコープ
} finally {
    int d = 40;
    ...               dのスコープ
}
...
```

throwsによる例外処理

throws節

- ▶ そのメソッドで発生する可能性のある例外を記述する
 - ▶ 例外処理自体は、そのメソッドの呼び出し元に任せる

```
public void method() throws <例外型> {  
    例外が発生する可能性がある処理  
}
```

サンプルプログラム

```
public class ThrowsSample1 {  
    public static void main(String[] args) {  
        System.out.println("mainを開始します");  
        try {  
            sleep();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("mainを終了します");  
    }  
    static void sleep() throws InterruptedException {  
        System.out.println("メソッドを開始します");  
        Thread.sleep(1000);  
        System.out.println("メソッドを終了します");  
    }  
}
```

実行結果

```
C:\¥java>java ThrowsSample1  
mainを開始します  
メソッドを開始します  
(1秒間スリープ)  
メソッドを終了します  
mainを終了します
```

throwsの用途

- ▶ 右記のプログラムの
場合、例外発生の
有無をmainメソッド
側で判断できない
- ▶ 呼び出し元で例外発
生の有無を判断した
い場合は、throws節
が有効

```
public class ThrowsSample2 {  
    public static void main(String[] args) {  
        System.out.println("mainを開始します");  
        sleep();  
        System.out.println("mainを終了します");  
    }  
    static void sleep() {  
        System.out.println("メソッドを開始します");  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("メソッドを終了します");  
    }  
}
```

実行結果

```
C:\¥java>java ThrowsSample2  
mainを開始します  
メソッドを開始します  
(1秒間スリープ)  
メソッドを終了します  
mainを終了します
```


例外クラスの構造

チェック例外と非チェック例外

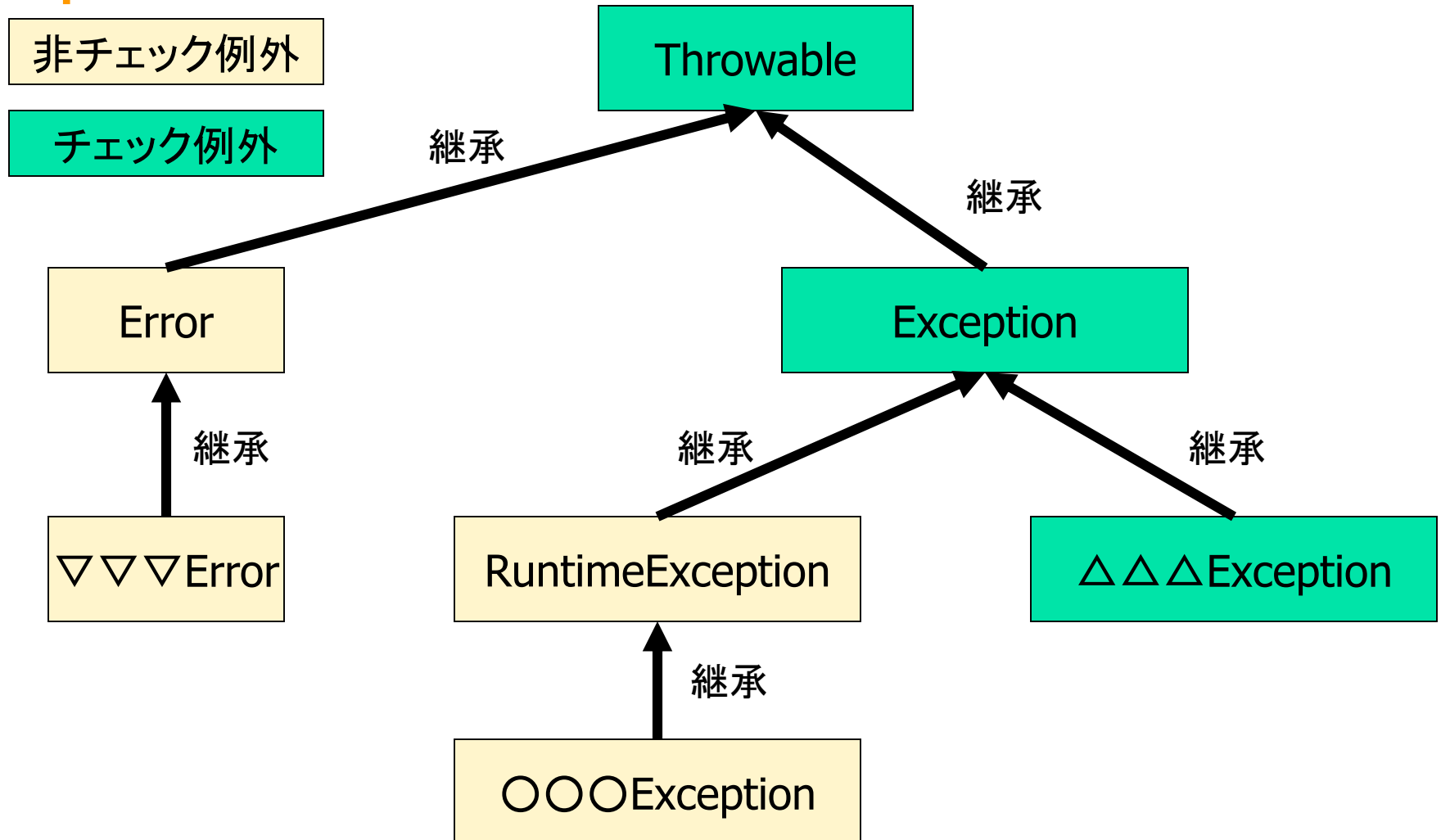
▶ チェック例外

- ▶ try-catchやthrowsなどの例外処理を記述していないと、コンパイルエラーになる例外

▶ 非チェック例外

- ▶ 例外処理を記述しなくても、コンパイルエラーにならない例外

例外クラスの構造



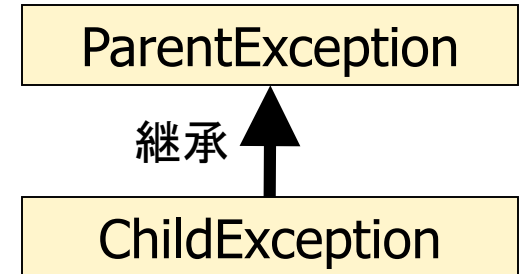
例外クラスの分類

分類	説明	チェック・非チェック	クラスの例
Throwable	全ての例外の親クラス	チェック	—
Errorとそのサブクラス	回復不能なエラーを表す	非チェック	OutOfMemoryError、NoClassDefFoundErrorなど
RuntimeExceptionとそのサブクラス	実行時例外を表す。多くはプログラミングのミスによって発生する	非チェック	ArrayIndexOutOfBoundsException、NullPointerException、NumberFormatExceptionなど
Exceptionとそのサブクラス (RuntimeExceptionとそのサブクラス以外)	主に外部環境が原因で発生する例外を表す	チェック	IOException、FileNotFoundException、SQLExceptionなど

親クラスによる例外キャッチ

▶ 親クラスでまとめてキャッチできる

▶ 例外の種類によって処理を変更できないので注意

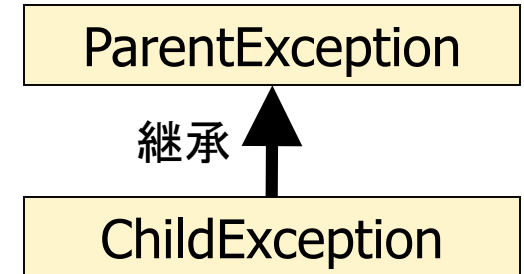


```
try {  
    // ParentExceptionが発生する処理  
    // ChildExceptionが発生する処理  
} catch (ChildException ce) {  
    例外処理  
} catch (ParentException pe) {  
    例外処理  
}
```

```
try {  
    // ParentExceptionが発生する処理  
    // ChildExceptionが発生する処理  
} catch (ParentException pe) {  
    例外処理  
}
```

catchの順序

◀ サブクラスから順に記述する



○

```
try {  
    //例外が発生する可能性がある処理  
} catch (ChildException ce) {  
    例外処理  
} catch (ParentException pe) {  
    例外処理  
}
```

×コンパイルエラー

```
try {  
    //例外が発生する可能性がある処理  
} catch (ParentException pe) {  
    例外処理  
} catch (ChildException ce) {  
    例外処理  
}
```

Java7の新機能

例外のマルチキャッチ

- ▶ 複数の例外に対する処理をまとめることができる
 - ▶ ○○Exception | △△Exceptionのように区切る
 - ▶ 処理が共通である例外にのみ利用すべき

```
try {  
    //例外が発生する可能性がある処理  
} catch (FooException | BarException e) {  
    共通の例外処理  
}
```


try-with-resources

- ▶ ファイル等のクローズ処理が、try-catchを抜けた際に自動で行われる
 - ▶ AutoCloseableインターフェイスを実装したクラスに限られる

```
try ( FileReader fr = new FileReader("hoge.txt");  
    BufferedReader br = new BufferedReader(fr) ) {  
    ...  
} catch (IOException e) {  
    ...  
}
```

↑
frとbrは自動で
クローズされる

その他のトピックス

例外のスロー

▶ throw new ○○Exception() で自分で例外をスロー出来る

```
...  
if (flag == false) {  
    throw new FooException();  
}  
...
```

例外の自作

- ▶ ○○Exceptionクラスを継承して作成
 - ▶ コンストラクタだけ定義することが多い

```
public class MyException extends Exception {  
    public MyException() {  
        super();  
    }  
    public MyException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```

例外処理のタブー

▶ Exceptionで例外をまとめてキャッチできる

```
try {  
    //例外が発生する可能性がある処理  
} catch (Exception e) {  
    例外処理  
}
```

▶ catch内に処理を記述しない

```
try {  
    //例外が発生する可能性がある処理  
} catch (Exception e) {  
    // 何も記述しない  
}
```

実務における例外処理

- ▶ プロジェクトによって、例外の構成や処理方法の基準が違うことがあるので、基本的にはプロジェクトの規約に従うこと

第9章

オブジェクト指向プログラミング のポイント

保守性と再利用性

▼ 保守性

- ▼ プログラムを修正しやすくする

▼ 再利用性

- ▼ プログラムを他のシステムでも使い回せるようにする

単一責任の原則

- ▶ 1つのクラスには1つの役割のみを持たせる
 - ▶ 1つのクラスに様々な役割を詰め込むと、保守性・再利用性が下がる