

## 2.2 繙承とポリモーフィズム

- 繙承
- 繙承の文法
- ポリモーフィズム
- インタフェース



# 目 次

1

継承

2

継承の文法

3

ポリモーフィズム

4

インターフェース

## Example ✓

動物園のシステムを設計したい。動物園には、猫や犬、ライオンや虎など、いろいろな動物がいます。オブジェクト指向の概念を用いて、猫クラスや犬クラスとかの色々なクラスを書き、それぞれのオブジェクトを別々に作成することができます。

- この際、以下の質問について考えてみなさい:
  - 猫や犬など、それぞれの動物には、**同じような性質**（属性または振る舞い）があるのでしょうか？
  - 異なる動物クラスを書くときに**重複したコード**を書くことはありますか？
  - 今まで習った文法で、これらの問題が解決できますか？

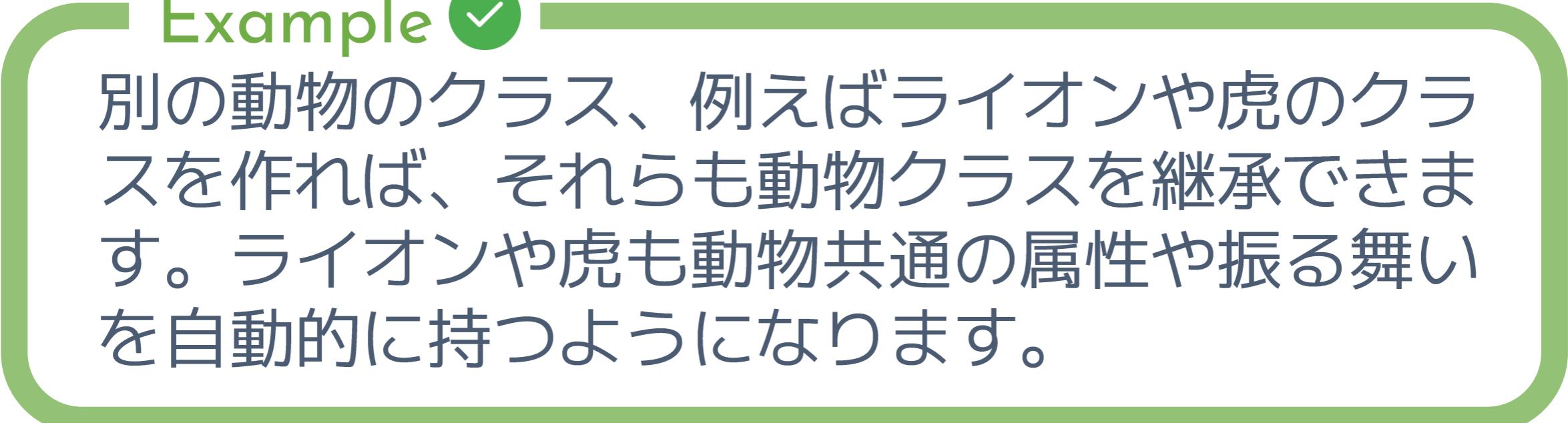
# 継承

- 猫と犬には異なる性質がありますが、共通するの性質もたくさんあります。例えば、身長、性別、年齢などの属性や、食べるや、寝るなどの振る舞いが、どの動物にもあります。
- つまり、このようにまとめることができます：どんな**動物**にも身長、性別、年齢などの属性がありますし、どんな**動物**も食事や睡眠などの行動があります。そして、**猫**も**犬**も、一種の**動物**であるゆえに、以上の性質を持ちます。

次へ 

 前へ

- 繙承[Inheritance]とは、「...は一種の... (is-a)」という関係性を意味する。まず、動物クラスを定義し、猫と犬にこの動物クラスを継承させることができます。これによって、猫や犬は、自動的に動物クラスの色々な性質を持つようになります。

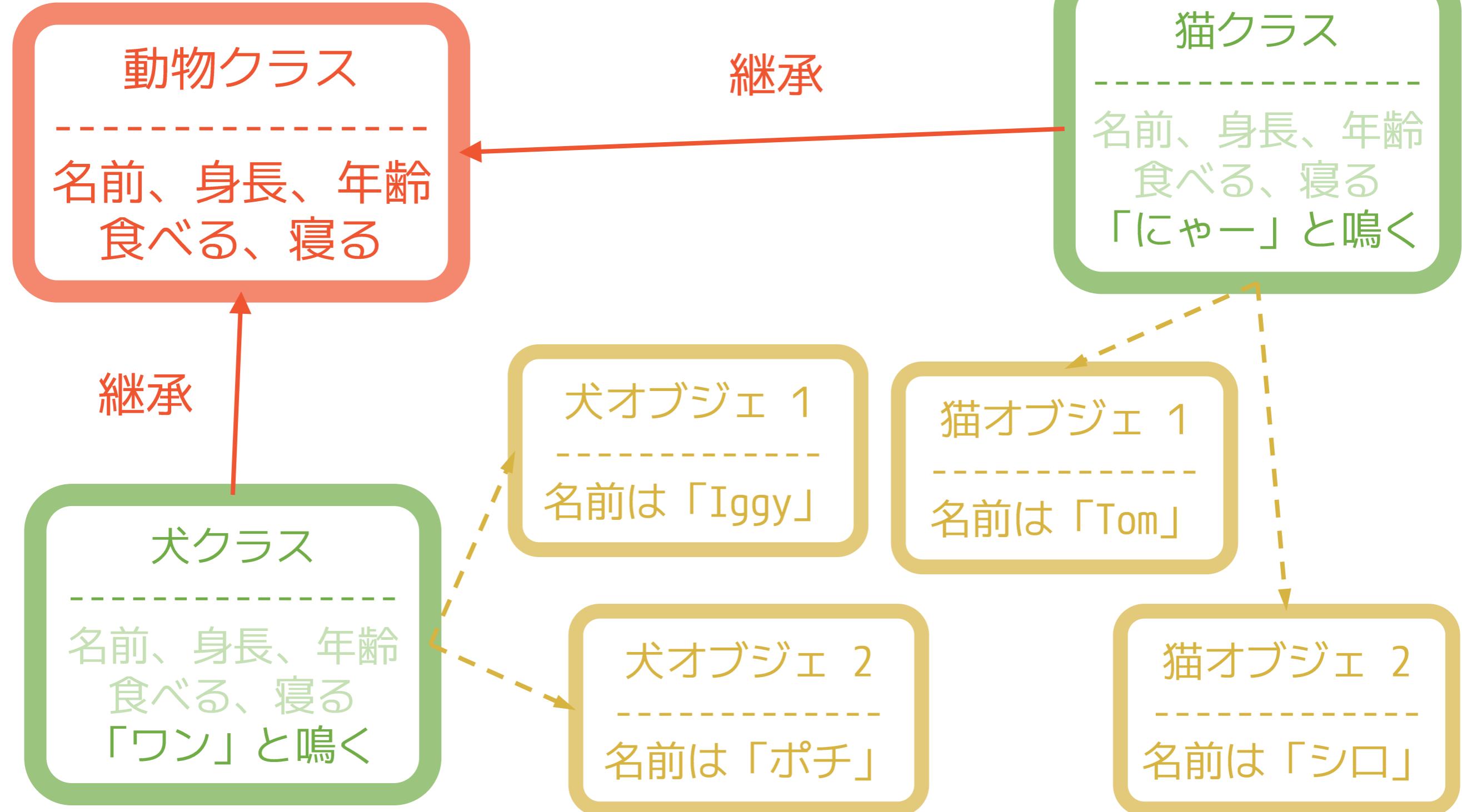


### Example

別の動物のクラス、例えばライオンや虎のクラスを作れば、それらも動物クラスを継承できます。ライオンや虎も動物共通の属性や振る舞いを自動的に持つようになります。

次へ 

◀ 前へ



# スーパークラスとサブクラス

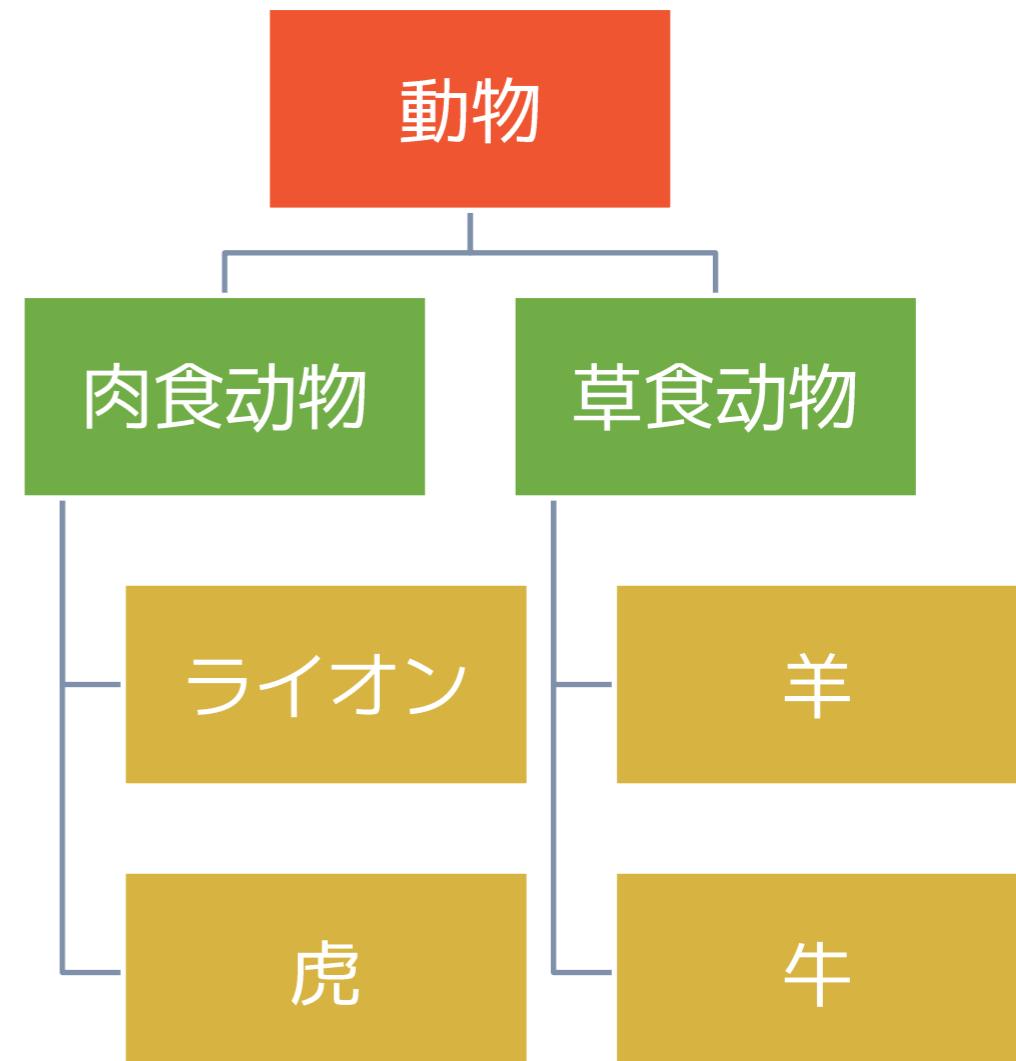
- クラス A がクラス B を継承する場合は、A を B のサブクラスまたは子クラス [Subclass] といい、B を A のスーパークラスまたは親クラス [Superclass] といいます。
- 継承関係の簡易判定方法：A が一種の B (A is a B) である場合、A は B を継承し、A は B のサブクラスであります。
- 以下のクラスのペアが、それぞれどのような継承関係を持っていますか？
  - 学生・大学生；
  - 柴犬・犬；
  - 正方形・長方形；
  - 生物・草食動物；
  - 人・植物。

Tips 

サブクラスのインスタンスは、スーパークラスのインスタンスでもある。

# 継承の階層

- クラスの間の継承関係は、階層的な構造になることが非常に多いです。例えば、ライオンや虎は肉食動物、羊や牛は草食動物、草食動物や肉食動物は動物：
- 階層的な継承は、推移性を満たすことにも注目：ライオンも羊も動物であります。
- この図の中、どのクラスがどのクラスの親、どのクラスがどのクラスの子にあたるかを考えてみてください。





*Question and answer*



1

継承

2

継承の文法

3

ポリモーフィズム

4

インターフェース

# Java での継承

- Java でクラスを定義する場合、**extends** キーワードを使ってそのスーパークラスを指定することができます：

```
1 public class Cat extends Animal {  
2     codes;  
3 }
```

- ここで、Cat クラスは Animal クラスを継承して、Animal クラスのサブクラスになりました。

次へ 

◀ 前へ

- 繙承することによって、スーパークラスの属性やメソッドのほとんどは、サブクラスで直接使用することができます：

Animal.java:

```
1 public class Animal {  
2     String name = "";  
3  
4     void eat(String food) {  
5         System.out.println(name + " eat " + food);  
6     }  
7 }
```

Cat.java:

```
public class Cat extends Animal { }
```

Main.java:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         Cat kitty = new Cat();  
4         kitty.name = "Kitty";  
5         kitty.eat("fish"); // => "Kitty eat fish"  
6     }  
7 }
```

次へ ▶

◀ 前へ

- また、サブクラスの特有なメンバ変数やメソッドを定義することも可能です：

Cat.java:

```
1 public class Cat extends Animal {  
2     void meow() {  
3         System.out.println("meow~");  
4     }  
5 }
```

Main.java:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         Cat kitty = new Cat();  
4         kitty.name = "Kitty";  
5         kitty.meow(); // => meow~  
6     }  
7 }
```

Note !

Javaでは、クラスは1つのスーパークラスしかを持つことができません。

# サブクラスのコンストラクタ

- サブクラスのコンストラクタの最初に、**super** を使ってスーパークラスのコンストラクタを呼び出すことができます。書かなかった場合は、自動的に「super()」を呼びだすことになります。

Animal.java:

```

1 public class Animal {
2     String name;
3
4     Animal(String name_) {
5         name = name_;
6     }
7 }
```

Main.java:

```

Cat kitty = new Cat("Kitty");
System.out.println(kitty.name); // => Kitty
```

Cat.java:

```

1 public class Cat extends Animal {
2     Cat(String name_) {
3         super(name_);
4     }
5 }
```

Try  
animals  
パッケージ

# サブクラスのデフォルトコンストラクタ

- 前に ( ↪ § 2.1.2) 、 **デフォルトコンストラクタ**について学びました。なお、サブクラスのデフォルトコンストラクタは、最初の行で自動的にスーパークラスの**引数なしのコンストラクタ**「super()」を呼び出します。
- さて、スーパークラスが引数のコンストラクタを持っていない場合、サブクラスの動きはどうなりますか？

Try  
01011  
11010  
01011

animals.constructor パッケージ

- スーパークラスが引数なしのコンストラクタを持たない場合、**サブクラスのコンストラクタを定義しなければならず、その最初に super() を書かなければなりません。**



*Question and answer*



# 目 次

1

## 継承

2

## 継承の文法

3

## ポリモーフィズム

4

## インターフェース

# メソッドのオーバーライド

- サブクラスでスーパークラスと同じ型（名前、引数の型、戻り値が全て同じ）で定義するメソッドは、スーパークラスのメソッドをオーバーライド[Override]すると言います。

## Example ✓

猫クラスと犬クラスは、どちらも動物クラスを継承しています。動物クラスには、食べるという動作を表すメソッドがあります。しかし、猫クラスは食後に「ニヤー」と鳴き、犬クラスは「ワン」と鳴くように、動物クラスで定義された動作とは異なる動作をします。したがって、このメソッドをオーバーライドして、新しい動きを実装することができます。

# オーバーライドとオーバロード

- オーバーライドとオーバロード<sup>[Overload]</sup>との違いに注意してください: オーバロードは、**引数リストが異なる**型であることを保証しなければなりません。これに対し、オーバーライドでは、**引数の型と戻り値の型の両方が同じ**であることを保証する必要があります。
- これら 2 つの単語は見た目も似ているし、意味も共通する点があるから、くれぐれも気を付けてください。一番大きな違いは、オーバロードは新しいメソッドを作りますが、オーバーライドは元々あるメソッドを書き替えだけです。

# ポリモーフィズム

- オーバーライドによって、同じクラスのオブジェクトで同じメソッドを呼び出しても動きが異なることがある：具体的な動きは、オブジェクトがどのサブクラスに属するかに依存します。この性質**はポリモーフィズム**[Polymorphism]と呼ばれます。

## Example ✓

猫クラスと犬クラスはともに動物クラスを継承し、「食べる」メソッドを異なるコードでオーバーライドしています。猫と犬をそれぞれ1匹作って、どちらも動物のオブジェクトとして扱うことができます。このとき、この2匹の「食べる」という動作の表現は違います。

次へ ↗

◀ 前へ

- 先ほどの例を Java で表すと: 犬や猫が動物なら、犬や猫のオブジェクト（インスタンス）もまた動物のオブジェクトです。なので、動物型の変数に入れることができるはず:

```
Animal kitty = new Cat("Kitty");
Animal iggy = new Dog("Iggy");
```

- さて、「kitty」というオブジェクトのどんなメソッドが使えるのでしょうか?

次へ ➞

 前へ

- Java は kitty が Cat であることは知りませんが、Animal であることは知っています。そこで、kitty は Animal が持つメソッドだけ呼び出すことができます：

```
kitty.eat("cat food"); // kitty は、動物クラスのメソッドが呼び出し可能  
kitty.meow(); // エラー: kitty は猫ということはわかっていない
```

- eat() メソッドのオーバーライドと合わせて、同じ Animal のオブジェクトでも、eat() を呼び出したときの挙動が異なります。

Try 

```
animals.polymorphism  
パッケージ
```

# ポリモーフィズムの役割

- ポリモーフィズムの実用的な価値は一体何なのでしょうか。

## Example ✓

動物園のシステムを考えてみましょう。動物園には何十、何百の**動物**がいて、別々のクラスを作る必要があります。

しかし、すべての**動物**のオブジェクトを、一つの配列を保持しておいて、統一に操作したい。

では、その配列のタイプは何にすればよいのでしょうか。

**猫**? **犬**? それとも.....

次へ ➞

◀ 前へ

- Java はオーバーライドでポリモーフィズムを実装できますので、すべてのオブジェクトを Animal 型の配列に統一することができます。すべての動物に順番にある動作をさせたい場合は、この配列を繰り返すすればいいです。

Try  
animals.zoo  
パッケージ

# アノテーション

- アノテーション[Annotation]は、Java の特殊な構文であります：コメントに似てて、Java コードとして実行されませんが、コードを処理する際に Java コンパイラや他のプログラムが**特別な操作**を行うことができます。
- アノテーションはすべて「@」で始まります。今は、以下の「**@Override**」というアノテーションを覚えらればいいです。

```
1 @Override  
2 void eat(String food) {  
3     System.out.print(name + " ate " + food + ", ");  
4     meow();  
5 }
```

# オーバーライドのアノテーション

- 「`@Override`」のアノテーションは、その後に続くメソッドがスーパークラスのメソッドをオーバーライドすることを宣言しています。もし、コンパイラがそのメソッドがスーパークラスにないことに気づいたら、**構文エラー**を報告します。
- アノテーションがなくてもオーバーライドは可能ですが、オーバーライドするメソッドの前に常にアノテーションを付けることで、コードのタイプミスを防ぎ、可読性を高めることができます。

# オーバーライドの特殊な使い方: `toString()`

- Java では、すべてのクラスは暗黙に基本クラスである「**Object**」を継承しています。
- **Object** で定義されたメソッドのいくつかをオーバーライドして、特殊な機能を実装することができる。例えば、**toString()** メソッドがオーバーライドできます:

```
@Override  
public String toString() {  
    return "I'm a cat named " + name;  
}
```



- これにより、複雑なオブジェクトを統一した形で文字列に変換して出力することが簡単になりました。例えば、`System.out.println()` メソッドでオブジェクトを直接に出力することもできます。



*Question and answer*



# 目 次

1

## 継承

2

## 継承の文法

3

## ポリモーフィズム

4

## インターフェース

# ポリモーフィズムの実装

## Example ✓

ポリモーフィズムにより、すべての動物を同じ配列にして、統一な処理を行うことができました。今は、動物園のシステムをさらに拡張し、動物だけでなく、人や車など、動物園にある他のものも扱えるようにしたい。

ここでもポリモーフィズムを利用できます：動物や車、人が「走る」ことができます。このような走れるものを一箇所に集め、その「走る」メソッドで事故時の避難をしたい。

動物、車、人を同じ配列にするには？

# 解決策

## Example ✓

方法 1: 「走るもの」を表すクラスを定義し、走れる動物、車、人にこのクラスを継承させます。

問題: Java では、多重継承はできません。犬は「動物」と「走るもの」を両方継承できません。

方法 2: 「走るもの」を表すクラスを定義し、そのクラスを動物クラス (Animal) 、車、人に継承させます。

問題: 魚は?

- この問題を解決するために、インターフェースを使用します。

# インターフェース

- インタフェース[Interface]は、特殊な Java クラスと理解することができます。通常の Java クラスと異なるのは:
  1. インタフェースにメンバ変数がない。
  2. インタフェースは**メソッドの定義のみ**を記述する。具体的な実装のコードはない(「{}」の部分がない)。このようなメソッドは、**抽象メソッド**[Abstract Method]と呼ばれる。
  3. インタフェースはインスタンス化できない。
- インタフェースを定義するには、**interface** キーワードを使います:

```
1 public interface Addable {  
2     int add(int a, int b);  
3 }
```
- ここでの add() は抽象メソッドで、その引数と戻り値のみが定義され、その実装方法は定義されていません。

# インターフェースの実装

- クラスを定義するとき、**implements** キーワードを使用し、そのクラスがあるインターフェースを実装することを示すことができます。これはクラス継承にやや似ている：このクラスは自動的にインターフェースの全メソッドの定義を持ちます：

```
public class Adder implements Addable { }
```

- ただし、抽象メソッドは未実装のままでです。クラスを実際に使う前に、これらのメソッドを実装するコードを書く必要があります（add() メソッドの前の public は後述します）：

```
public class Adder implements Addable {  
    @Override  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

# インターフェースの使用

- インタフェースの実装がクラスの継承と似ているのは、インターフェース A を実装したクラス B のインスタンスも A のインスタンスになります。つまり、**クラス B のインスタンスを格納**するために、インターフェースタイプ A の**変数**を使うことができます。

```
1 Addable addable = new Adder();
2 System.out.println(addable.add(1, 2)); // => 3
```

- インターフェイスには、クラスとは異なるもう一つの大きな特徴がある。それは、一つのクラスは**複数**のインターフェースを**実装**できることです。

次へ ↗

 前へ

- 複数のインターフェースを実装する構文は以下の通り：

```
public class Cat extends Animal implements Runnable, Eatable { }
```

### Example ✓

「走るもの」を表すインターフェースとして  
Runnable を定義し、走る動物、車、人にこの  
インターフェースを実装させることができます。  
動物、車、人は他のスーパークラスを継承して  
も構いません。

Try   
animals.bigzoo パッケージ

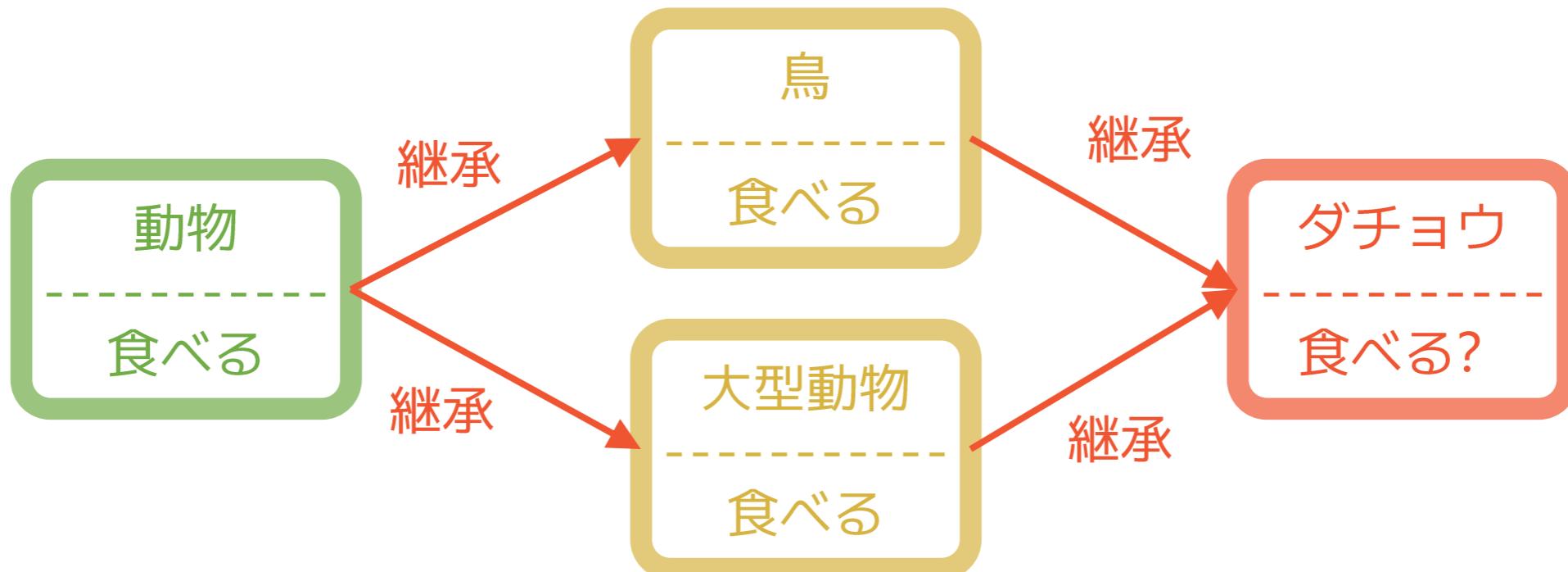


*Question and answer*

## Coffee ☕ Break

## 多重継承と菱形継承問題（1）

なぜ、Javaには多重継承はありませんか。次のようなシナリオを考えてみましょう：



Javaがダチョウのオブジェクトの「食べる」のメソッドを呼び出したいとき、一体「鳥」と「大型動物」のどのクラスのコードを実行したらいいですか？

## Coffee ☕ Break

## 多重継承と菱形継承問題（2）

C++ のように多重継承をサポートするプログラミング言語では、このような**菱形継承問題**[Diamond Problem]によってコードの論理構造が混乱し、可読性が低下して開発効率に影響を与える恐れがあります。同時に、これらの言語のコンパイラは開発が難しく、構文自体がより複雑になることもあります。

Java では、インターフェースを使って疑似的な多重継承をすることでこの問題を回避します。このインターフェースを使った仕組は実践において、開発効率と可読性のバランスが非常に良いので、他の多くの言語もこの仕組みを参考しています。

# まとめ

## Sum Up

1. 繙承の基本概念。
2. Java の継承構文：
  - ① サブクラスの定義と使い方。
  - ② サブクラスのコンストラクタ。
3. ポリモーフィズム：
  - ① ポリモーフィズムの基本概念。
  - ② ポリモーフィズムの実装方法：オーバーライド。
  - ③ インタフェース。



*Question and answer*

**THANK YOU!**