

## 2.4 オブジェクト指向補足

- 抽象クラス
- this と super
- オブジェクト参照
- 列挙型
- 内部クラス

# 目次

- 1 抽象クラス
- 2 this と super
- 3 オブジェクト参照
- 4 列挙型
- 5 内部クラス

# 抽象クラス

- **抽象クラス**<sup>[Abstract Class]</sup>とは、特殊のクラス的一种で、インタフェースと同様に、実装されていないメソッド（**抽象メソッド**<sup>[Abstract Method]</sup>）を含みます。インスタンスと同様に、**インスタンス化もできません**。ただし、実装されているメソッドやメンバ変数を含むことができます。
- 抽象クラスを定義するには、**abstract** 修飾子を使います：

```
public abstract class Animal { }
```

- メンバ変数や実装メソッドについては、通常のクラスと同じように定義します。**未実装のメソッド**の場合も、**abstract**修飾子を使用します：

```
public abstract void eat(String food);
```



# 抽象クラスの使用

- インターフェースと同様に、抽象クラスは自らインスタンス化できません。抽象クラスを継承し、抽象メソッドを実装した**サブクラス**を定義する必要があります：

```

1 public class Cat extends Animal {
2     @Override
3     public void eat(String food) {
4         System.out.print(getName() + " ate " + food + ", ");
5         meow();
6     }
7 }

```

- 簡単に言うと、抽象クラスは、誤ってインスタンス化されることを防ぐために使われている特殊なクラスだと認識すればいいです。

Try  animals パッケージ

# インタフェースと抽象クラスの継承

- 一つのクラスは任意の数のインタフェースを継承することができます：

```
public interface InterfaceA extends InterfaceB, InterfaceC { }
```

- InterfaceA をインスタンス化したクラスは、InterfaceB と InterfaceC のメソッドも実装する必要があります。
- 抽象クラスが何らかのインタフェースを実装したり、抽象クラスを継承したりする場合、そのインタフェースやスーパークラスの中にある未実装メソッドは未実装のまま、**サブクラスに実装してもらう**ことができます：

```
public abstract class Animal implements Runnable { } // エラーなし
```

- インタフェースとは違い、抽象クラスは本質的にクラスであるため、**多重継承**されることは**できません**。

# Q&A

# 目次

- 1 抽象クラス
- 2 this と super
- 3 オブジェクト参照
- 4 列挙型
- 5 内部クラス

# this キーワード

- Java における this キーワードは、**現在のオブジェクト**を指す場合と、コンストラクタにおいて**他のコンストラクタ**を指す場合の 2 つの意味があります。
- まず、this キーワードを一般の（静的じゃない）メソッドで使う場合に、**現在のオブジェクト**を意味します。this の後に「.」で変数名かメソッド名を繋げれば、このオブジェクトのメンバ変数やメソッドを使えます：

```
1 public void eat(String foog) {  
2     System.out.println(this.name + " ate " + food);  
3     this.meow( );  
4 }
```



# this で現在のオブジェクトを参照

- 前述のように、このクラスのメンバー変数やメソッドを直接使用する場合、実際には**現在のオブジェクト**の変数やメソッドを使用することになります。つまり、a はメンバ変数なら、「a」も「this.a」も**同じもの**を指します。
- では、this キーワードは一体何のために使うのでしょうか？
- this は、インスタンス変数とローカル変数の明確に**見分ける**ために使えます：

```
1 private void setName(String name) {
2     this.name = name;
3 }
```

# this でコンストラクタを参照

- this は、現在のクラスの**他のコンストラクタ**を参照するためにも使用することができます：

```
public Animal(String name) {
    this.name = name;
}

public Animal() {
    this("Unnamed Cat");
}
```

- これにより、通常の方法と同様に、引数のデフォルト値を簡単に設定することができます。

# super キーワード

- super キーワードは、親クラスを指し、継承関係にある際に使用する。superキーワードを使用することで、子クラスから、親クラスの**コンストラクタ**や**メソッド・メンバ変数**を呼び出すことができる。
- 主にスーパークラスの名前**メソッド**（オーバーライドされたメソッド）を呼び出すために使われます：

```
1 public void eat(String food) {
2     super.eat(food);
3     meow( );
4 }
```

- コンストラクタの用法については、この前（[← § 2.2.2](#)）すでに勉強しました。
- this と super を活用して、animals パッケージのコードを改善できますか。


# Q&A

# 目次

- 1 抽象クラス
- 2 this と super
- 3 オブジェクト参照
- 4 列挙型
- 5 内部クラス



# おさらい：基本データ型と参照型

- Java における型は、**基本データ**と**参照型**に分けられると（ § 1.2.1）述べました。その違いをおさらいしてみましょう。
  - 基本データ型は単純なデータ、参照型は複雑なデータを格納します。
  - 基本データ型のデータはオブジェクトではないのでメソッドを使用できませんが、参照型のデータはできます。
  - 基本データ型はコンストラクタを持ちません。
  - 基本データ型を継承することはできません。
- ここでは、両者の重要な違いである「格納の仕組み」をご紹介します。

# 基本データ型の代入

## Example

次のコードはどのような結果を出力すると思いますか？

コードを書いて、実行してみなさい。自分の予想と一致しますか？

```
1 int a = 1;
2 int b = 2;
3 a = b;
4 a = 3;
5 System.out.println("a = " + a);
6 System.out.println("b = " + b);
```

# 参照型の代入

## Example

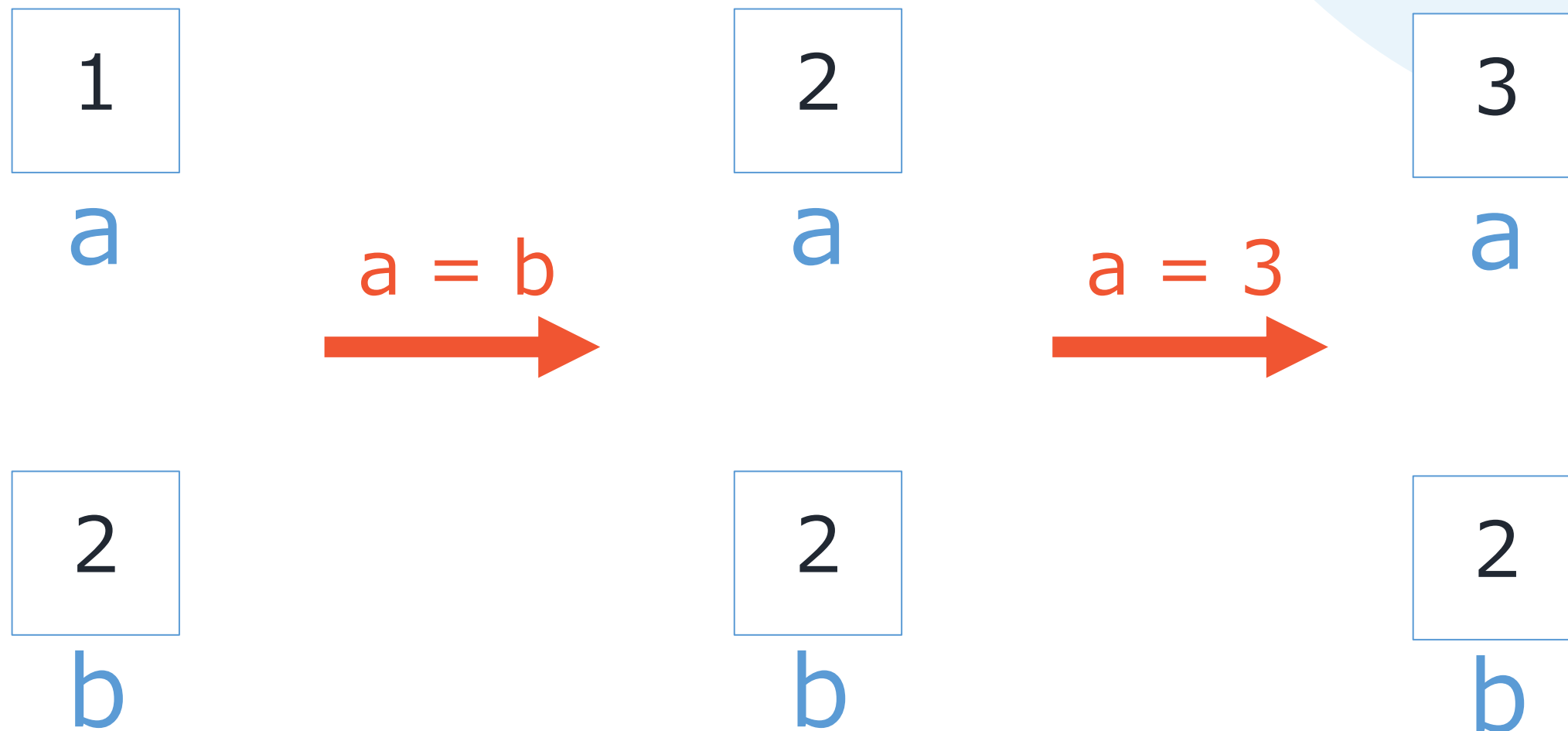
次のコードはどのような結果を出力すると思いますか？

コードを書いて、実行してみなさい。自分の予想と一致しますか？

```
1 int[] a = {1, 1};
2 int[] b = {2, 2};
3 a = b;
4 a[0] = 3;
5 System.out.println("a = " + a[0] + " " + a[1]);
6 System.out.println("b = " + b[0] + " " + b[1]);
```

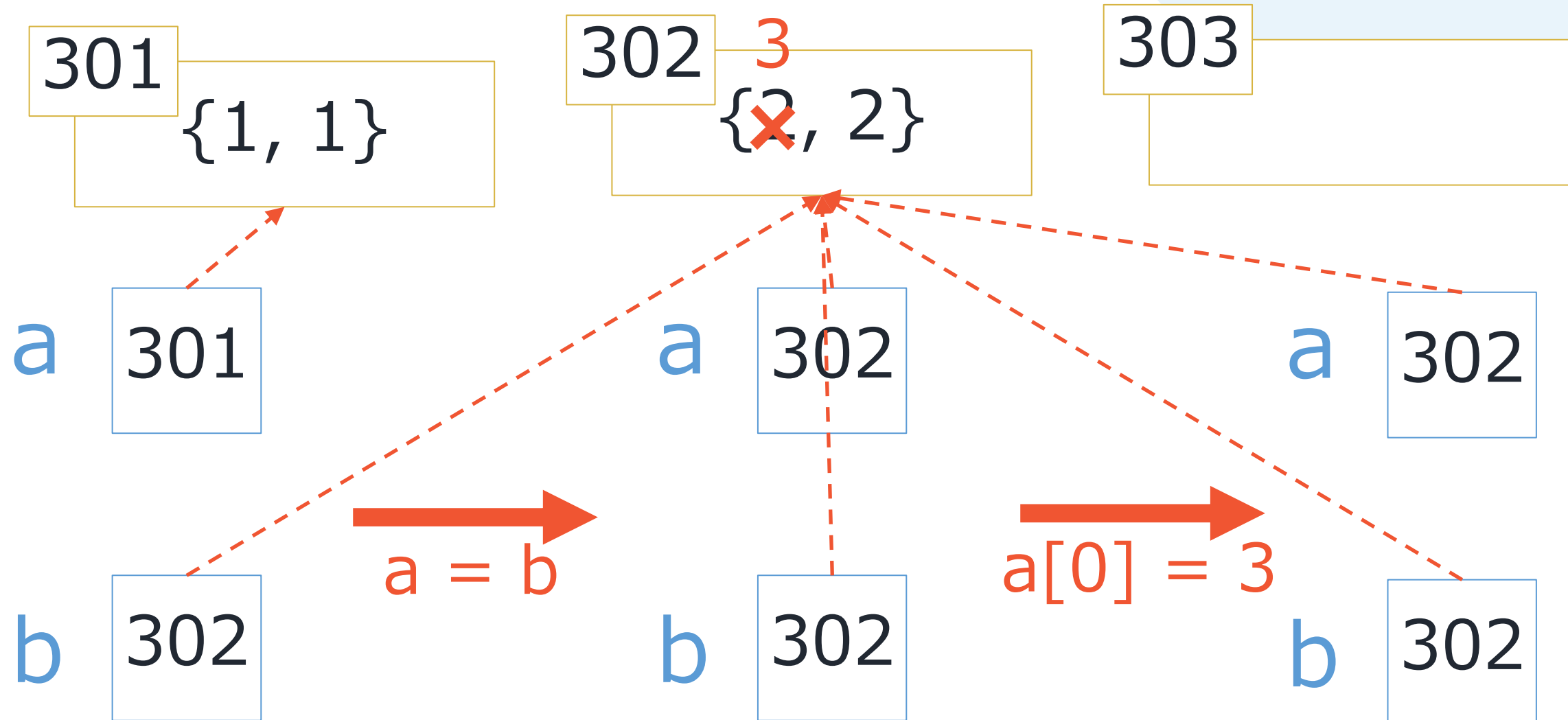
# 基本データ型を格納する方法

- Java では、基本データ型の変数は、データの値を直接格納します。代入文は、単に変数の値を**コピー**するだけです。したがって、2 つの変数の間は何の関係も**生じません**。



# 参照型を格納する方法

- 一方で、参照型変数には、オブジェクトへの**参照**<sup>[Reference]</sup>が格納されます。簡単に言うと、変数はオブジェクトの「部屋番号」を格納するようなものです。





# 自分で定義した参照型

- 当然ながら、自分で作ったクラスなどの参照型の変数にも、オブジェクトへの**参照**を格納しています。

## Example

```
1 Cat a = new Cat("Alice");  
2 Cat b = new Cat("Bob");  
3 a = b;  
4 a.setName("Charlie");  
5 System.out.println("a = " + a.getName()); // => a = Charlie  
6 System.out.println("b = " + b.getName()); // => b = Charlie
```

# 参照型変数の使用

- 一言で言えば、参照型変数間の直接代入は、変数間の「連携」をもたらし、**同じオブジェクト**を格納することになります。この操作を「**シャローコピー**[Shallow Copy]」と呼びます。
- したがって、参照型の変数を扱う場合は、代入記号「=」を慎重に使用する必要があります。例えば、配列をコピーしたい場合は、むやみに**代入記号を使ってはいけません**。

# Array のコピー

- 正しいコピー（**ディープコピー**<sup>[Deep Copy]</sup>）を行うためには、新しい配列を作成し、元の配列の**全要素**を順番に新しい配列にコピーする必要があります：

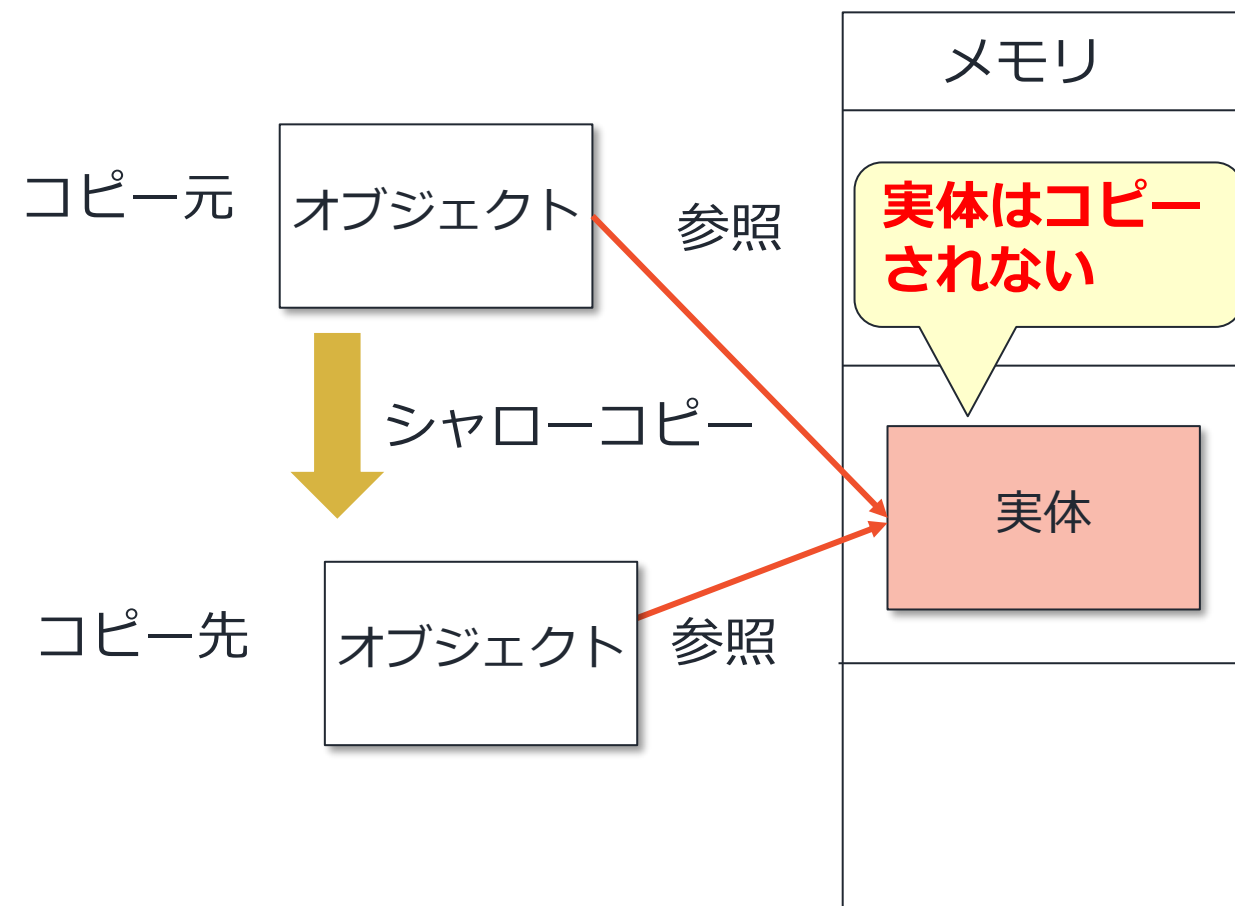
```
1 int a = {1, 1};
2 int b = {2, 2};
3 a = new int[2];
4 for (int i = 0; i < 2; i++) {
5     a[i] = b[i];
6 }
7 a[0] = 3;
8 System.out.println("a = " + a[0] + " " + a[1]); // => a = 3 2
9 System.out.println("b = " + b[0] + " " + b[1]); // => b = 2 2
```

- ちなみに、Java では Array クラスに、配列をコピーするためのメソッド **clone()** も用意されています：

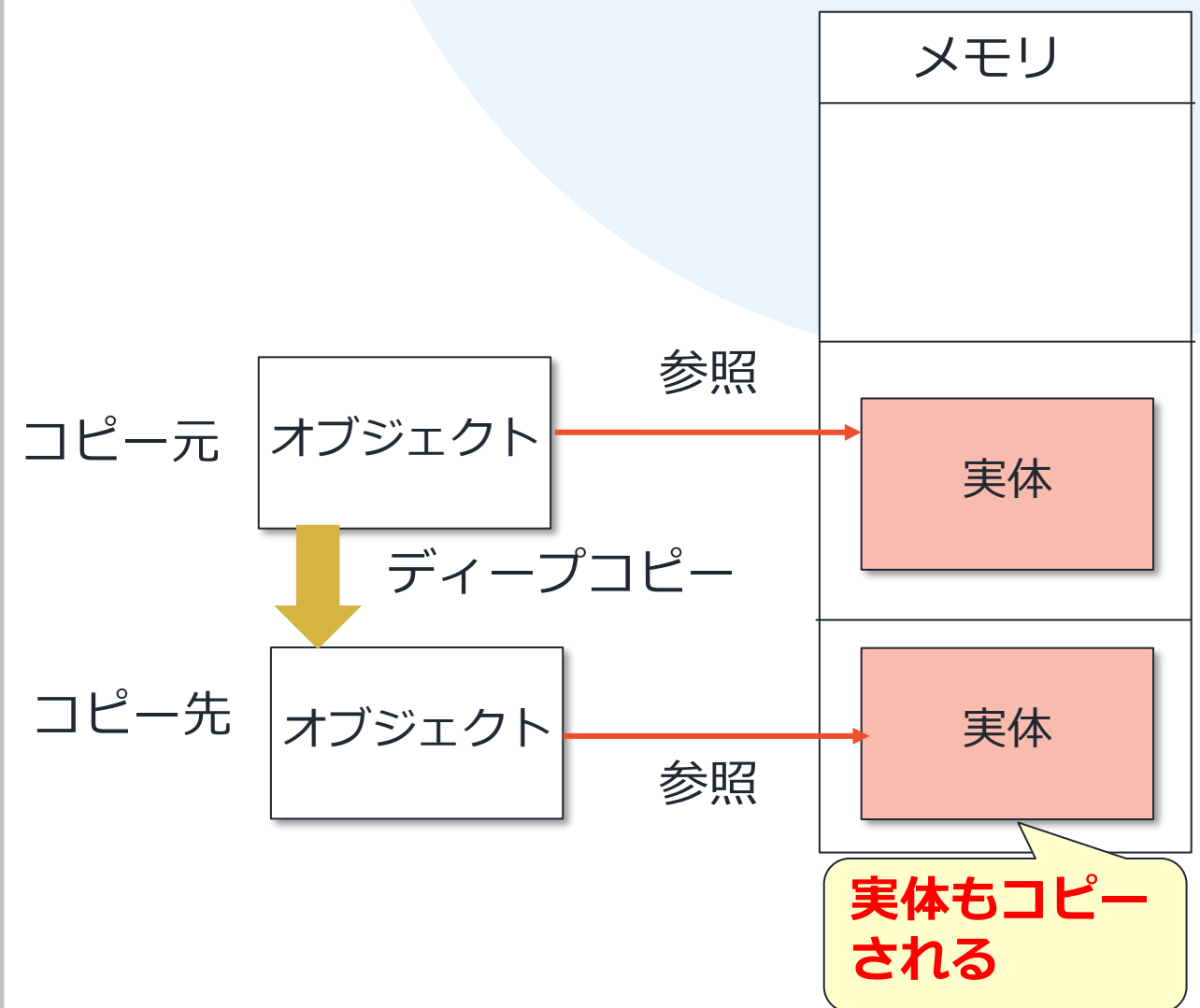
```
a = b.clone();
```

# シャローコピーとディープコピー

## シャローコピー



## ディープコピー



## 2 次元配列のコピー

- 考えてみよう：**2 次元の配列**を clone() メソッドで正しくコピーできるのでしょうか？
- ダメならば、それはなぜでしょうか？そして、正しくコピーするコードを書けますか？

Try

```
01011
11010
01011
```

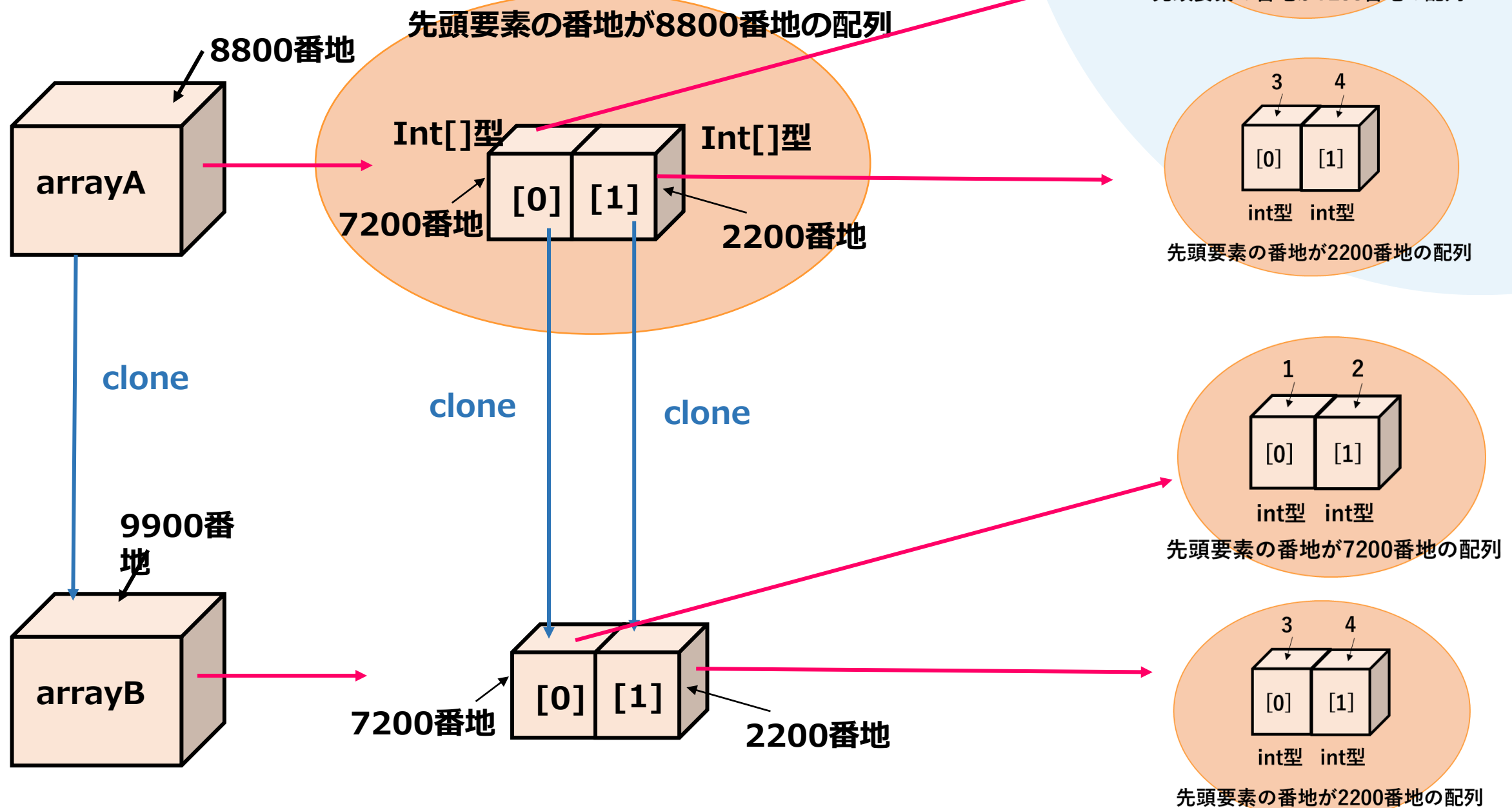
TwoDCopy.java

- まだ理解できなければ、**2 次元配列をコピーするときにこのコードを使おう！**さえ覚えれば大丈夫でしょう。



# 2 次元配列のコピー図

```
int[][] arrayA = { { 1, 2 }, { 3, 4 } };
int[][] arrayB = arrayA.clone();
```



# イコール演算子「==」

- Java におけるイコール演算子「==」は、2 つの変数が等しいかどうかを判定できることが分かっています。しかし、**参照型**にイコール演算子を使う場合は、十分に注意する必要があります。
- 実際に比較されたのは、オブジェクトへの**参照**になります：

```
int[] a = {1,2}; // => アドレス:[I@5ca881b5
int[] b = {1,2}; // => アドレス[I@24d46ca6
System.out.println(a == b); // => false
```

- この例では、a と b は**異なる**配列ですから、たまたま同じ値を含んでも、「a == b」は「false」を返します。

# Stringの特徴

- Stringは、**定数**である  
・つまり変更できない
- データ自体は、メモリ上の別の場所に置かれて  
そのアドレスの値 **参照値** が変数に入る。
- 中に入っている値は「**equalsメソッド**」  
参照している場所の比較は「**==**」で行う。

## クラスString

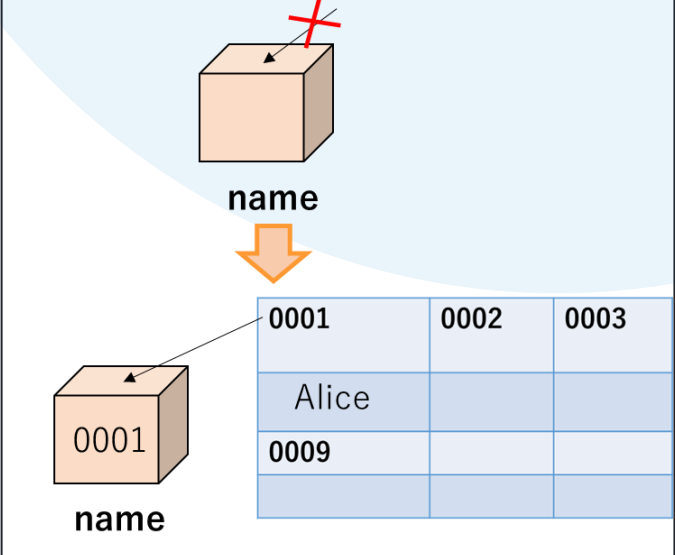
java.lang.Object  
java.lang.String  
すべての実装されたインタフェース:  
Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

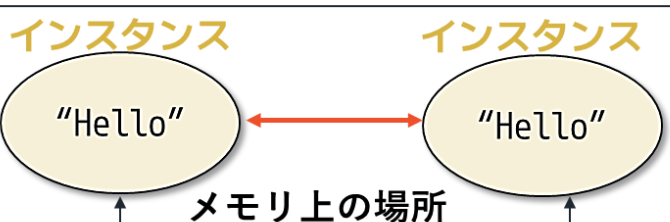
Stringクラスは文字列を表します。Javaプログラム内の"abc"などのリテラル文字列はすべて、このクラスのインスタンスとして実行されます。

文字列は定数です。この値を作成したあとに変更はできません。文字列バッファは可変文字列をサポートします。文字列オブジェクトは不変であるため、共用することができます。たとえば、

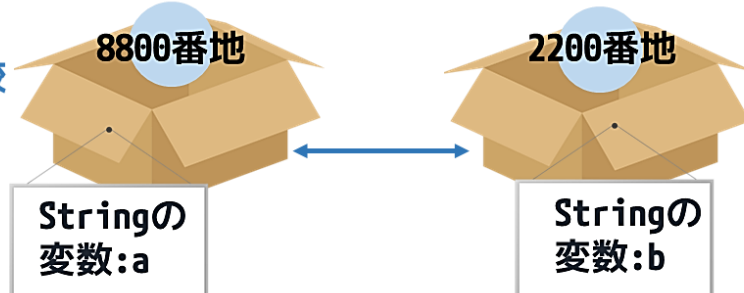
String name = "Alice";  
Alice



中に入っている値の比較:equalsメソッド



参照している場所の比較  
==, !=



```
String a = "Hello";
String b = new String("Hello");
```

# 文字列の比較

- 非常に間違えやすいのが、文字列 (String) を比較するときです：

```
1 String a = "abc";  
2 String b = "ab";  
3 b += "c";  
4 System.out.println(a == b); // => false
```

- 幸いに、Java は **equals()** メソッドを用意してくれたのです：

```
1 String a = "abc";  
2 String b = "ab";  
3 b += "c";  
4 System.out.println(a.equals(b)); // => true
```

## Note

文字列の比較には必ず **equals()** メソッドを使用すること！

# Q&A

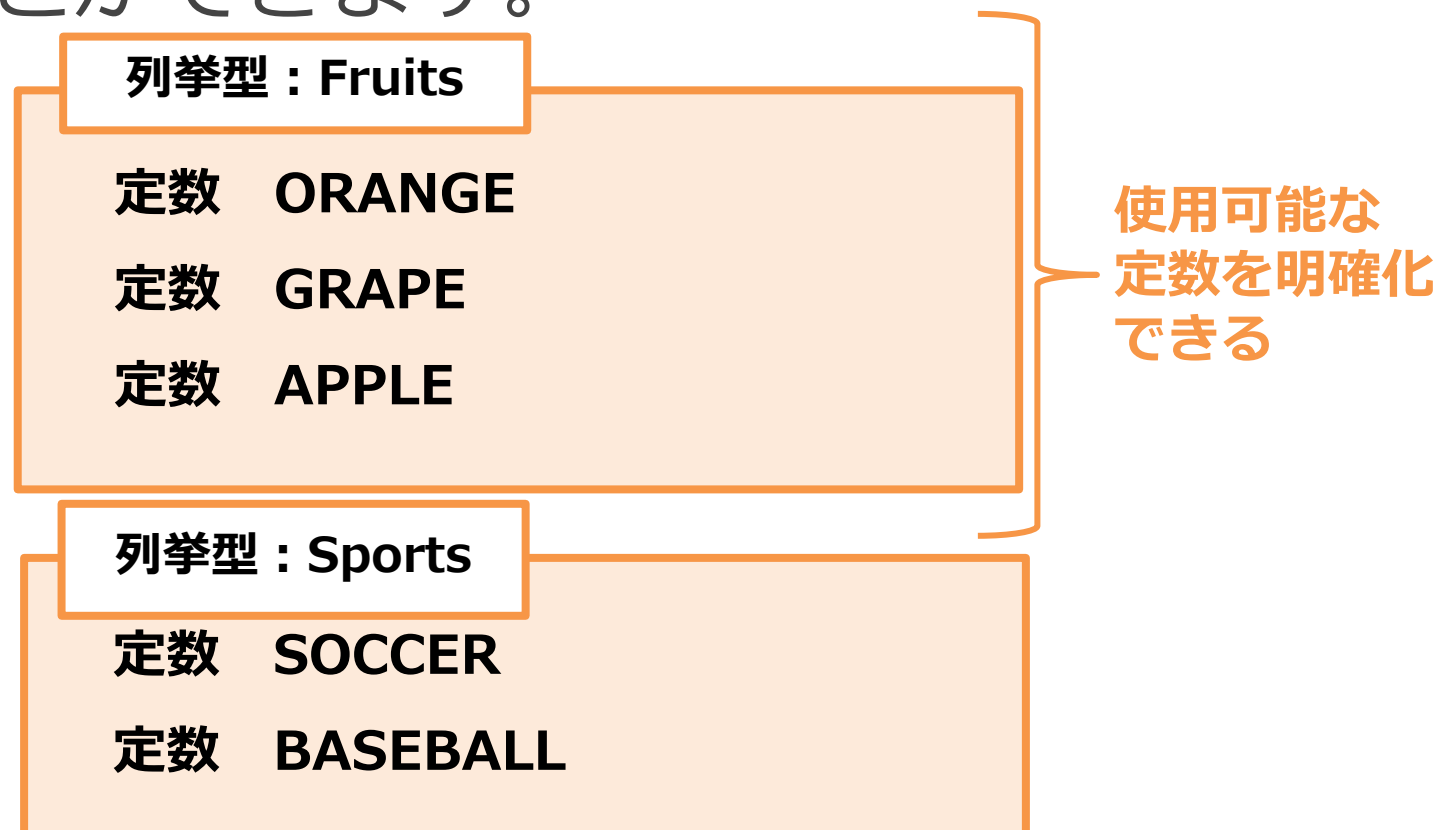


# 目次

- 1 抽象クラス
- 2 this と super
- 3 オブジェクト参照
- 4 列挙型
- 5 内部クラス

# 列挙型

- **列挙型**<sup>[Enum]</sup>とは、**関連する定数をひとつにまとめておくことができる型**のことです。列挙型で管理することで、**使用可能な定数を明確化**することができます。
- Enum（列挙型）で定義する定数のことを列挙子と言います。
- Javaの列挙型はクラスですので、フィールドやメソッドを定義することができます。



列挙型で管理する場合

# 列挙型の定義

- 列挙型を定義するには、**enum** キーワードを使用します：

```
1 public enum Status {  
2     RUNNING, POWERED_DOWN, SLEEPING  
3 }
```

- ここで、中括弧「{ }」には、取りうるすべての列挙子を書きます。列挙子のリストはカンマ「,」で区切ります。

Note



列挙子は、**大文字のスネークケース**で命名すべきです。

# 列挙型の使用

- 列挙型は他のクラスと同じよう扱うことができます。つまり、「Status」は普通のクラス名として、**変数や引数のタイプ**になり得ます。なお、Status 型の変数を取り得る値は、上記の 3 つだけです：

```
Status pc1Status = Status.RUNNING;
Status pc2Status = Status.POWERED_DOWN;
```

- 列挙された列挙子が、**クラス変数**（static な変数）と同じように使われていることがわかります。

# 列挙型と switch 文

- 列挙型のもう一つの便利な点は、**switch 文** (← § 1.3.1) で直接に扱うことができます：

```
1 switch (pc1Status) {  
2     case RUNNING:  
3         System.out.println("PC1 is running.");  
4         break;  
5     case POWERED_DOWN:  
6         System.out.println("PC1 is not running.");  
7         break;  
8     case SLEEPING:  
9         System.out.println("PC1 is sleeping.");  
10        break;  
11 }
```

# 列挙型のインスタンス変数とメソッド

- 列挙型はクラスなので、メンバ変数やメソッド、コンストラクタを持つこともできます。コンストラクタがある場合、定義された各列挙子に対して呼び出す必要があります。

```

1 public enum State {
2     RUNNING(1.0f), POWERED_DOWN(0.0f), SLEEPING(0.2f); // この「;」は省略不可
3
4     private float power;
5
6     State(float power) {
7         this.power = power;
8     }
9
10    public float getPower() {
11        return power;
12    }
13 }

```

Try  enums パッケージ

- 列挙型のコンストラクタは自動的に private になります。



# Q&A

# 目次

- 1 抽象クラス
- 2 this と super
- 3 オブジェクト参照
- 4 列挙型
- 5 内部クラス

# 内部クラス

- Java では、あるクラスの中に他のクラスを定義することができます。これは**入れ子クラス**や**内部クラス**[Inner Class]と呼ばれます：

```
1 public class Outer {
2     class Inner { }
3 }
```

- よくある使い方は、いくつかの小さなクラスを、それらを使うクラスに**集約される**ことです。例えば、**タイヤ**のクラスは、**車**のクラスだけに使われています。この場合、**タイヤ**クラスを**車**クラスの内部分類として書いてもいいでしょう：

```
1 public class Car {
2     private class Wheel { }
3 }
```

# 静的・非静的内部クラス

- 変数やメソッドと同様に、内部クラスも静的（static）と非静的の 2 種類があります。静的な内部クラスは直接インスタンス化することができますが、非静的な内部クラスは外部クラスのオブジェクトによってインスタンス化する必要があります。インスタンス化されたオブジェクトはこの外部クラスのオブジェクトに依存します。
- 一般的に、静的な内部クラスは単純な、あるいは誰でも使えるユーティリティクラスを実装するために使います。非静的な内部クラスは、外部クラスに依存するクラスを実装するために使います。
- 例えば、クラス内で使用する**列挙型**は、よく内部クラスとして記述されています。

# 静的内部クラス

- 静的内部クラスは、機能的には通常のクラスと変わりなく、使用する主な目的は、異なるクラスと一緒に**まとめる**ことです。static キーワードを使用して定義します：

```
1 public class Car {  
2     Wheel[] wheels;  
3  
4     private static class Wheel { }  
5 }
```

- 外部クラス以外の他のクラスを使用するには、**外部クラス名**と内部クラス名を「.」で繋いで使わなければなりません：

```
Car.Wheel wheel = new Car.Wheel();
```

# 非静的内部クラス

- 非静的内部クラスは、一般的に外部クラスに明示的に**依存する**クラスを定義するために使用されます。例えば、運転手は常に自分の車を運転することができます：

```
1 public class Car {  
2     private Driver Driver;  
3  
4     private class Driver { }  
5 }
```

- 他のクラスがこのような内部クラスをインスタンス化したい場合は、外部クラスの**オブジェクト名**と「new」を「.」で繋いで使う必要があります：

```
1 Car car = new Car();  
2 Car.Driver driver = car.new Driver();
```

次へ 

- 非静的内部クラスは、外部クラスのあるオブジェクトに依存しているため、そのオブジェクトのメンバ変数やメソッドを直接利用することができます：

```
1 public class Car {  
2     private Driver driver;  
3     private String brand;  
4  
5     private class Driver {  
6         public String getCarBrand() {  
7             return brand;  
8         }  
9     }  
10 }
```





# 内部クラス：まとめ

## Sum Up

内部クラスの構文や使い方は複雑ですが、現段階で覚えておきたいことは 2 つだけです：

1. 内部クラス名の表現方式、すなわち **Outer.Inner** の形式。なぜなら、後で使う外部ライブには内部クラスがあります。
2. 非静的内部クラスは、外部クラスのインスタンス変数やメソッドにアクセスすることができます。なぜなら、後では特別な非静的内部クラス（[➡ § 3.2.2](#)）を使うことになります。

# Q&A

# まとめ

Sum Up



1. 抽象クラスの定義と使用。
2. `this` と `super` キーワード：オブジェクトへの参照、コンストラクタへの参照。
3. オブジェクト参照：
  - ① `Array` のコピー：シャローコピーとディープコピー。
  - ② `String` の比較。
4. 列挙型の定義と使い方。
5. 内部クラスの基本的な使い方。



*Light in Your Career.*  
**THANK YOU!**