Woodland Academy

2.4 オブジェクト指向補足

- this と super
- オブジェクト参照

Shape Your Future

目次

thisとsuper メンバ変数とメソッドの呼び出し

- this.~ = 自分のオブジェクトの~
- super.~　=スーパー（親）クラスの~

```java
public class Animal {

    String name;

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

    public void eat(String food) {
        System.out.println(name + "は"+food+"を食べる");
    }
}
```

メンバ変数のnameを指している

```java
public class Cat extends Animal{

    public Cat(String name) {
        super(name);
    }


    public void sounds() {
        System.out.println("ニャ～");
    }

    @Override
    public void eat(String food) {
        super.eat(food);
        sounds();
    }

}
```

親クラスのメソッドを指している。

メリット
・同じコードを書かなくて良い
・引数名を考えなくて良い

Woodland Academy

thisとsuper コンストラクタの呼び出し

- this（引数）＝同じクラスの中でコンストラクタを呼び出しあう
- super（引数）＝スーパー（親）クラスのコンストラクタを呼び出す

```java
public class Animal {

    String name;

    public Animal() {
        this("未設定");
    }

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

    public void eat(String food) {
        System.out.println(name + "は"+food+"を食べる");
    }

}
```

文字列を代入する処理というのがすでに書かれているので、この下のAnimalのコンストラクタを利用しようということで、this(“未設定”)という書き方をしています

新しいコンストラクタをつくらなくても、すでに引数のあるコンストラクタを使って、nameに値を代入する

thisというのは、このクラスのという意味になりますので、このthisがAnimalに置き変わるイメージを持ってください。そうすると、同じクラスの中の引数1つのコンストラクタを呼び出すということで、この中で未設定が、this.nameの中に設定されるという動きになります。

```java
public class Cat extends Animal{

    String color;

    public Cat(String name, String color) {
        super(name);
        this.color = color;
    }

    public void sounds() {
        System.out.println("ニャ～");
    }

    @Override
    public void eat(String food) {
        super.eat(food);
        sounds();
    }

}
```

親クラスのコンストラクタを借りに行く

※this(~)、super(~)はコンストラクタ内の先頭に記述
（なければ自動的にsuper();が挿入される）

Woodland Academy

Q&A

目次

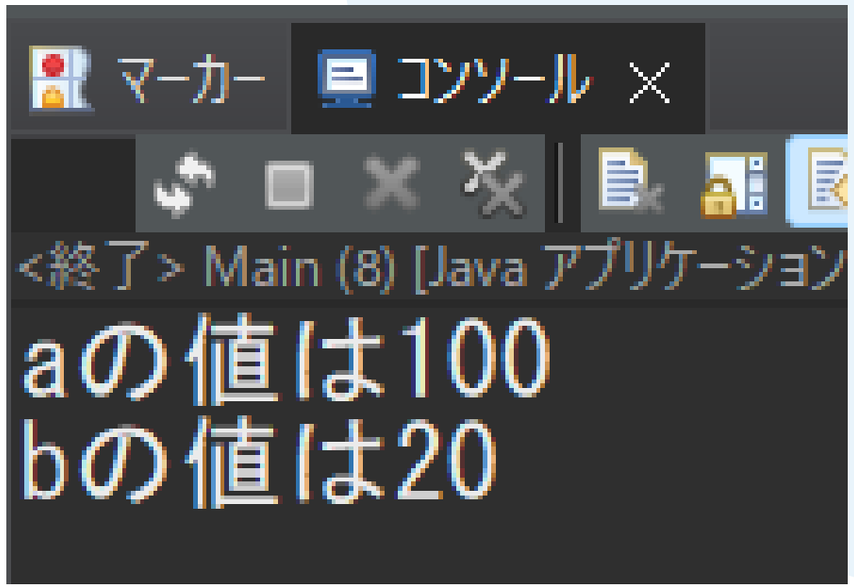
考えてみよう

- aの値とbの値は何になるかを考えてみてください

```
public class Main {

    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        //bの内容をaに代入
        a = b;
        //aに100を代入
        a=100;
        System.out.println("aの値は"+a);
        //bに20を加算
        System.out.println("bの値は"+b);
    }

}
```

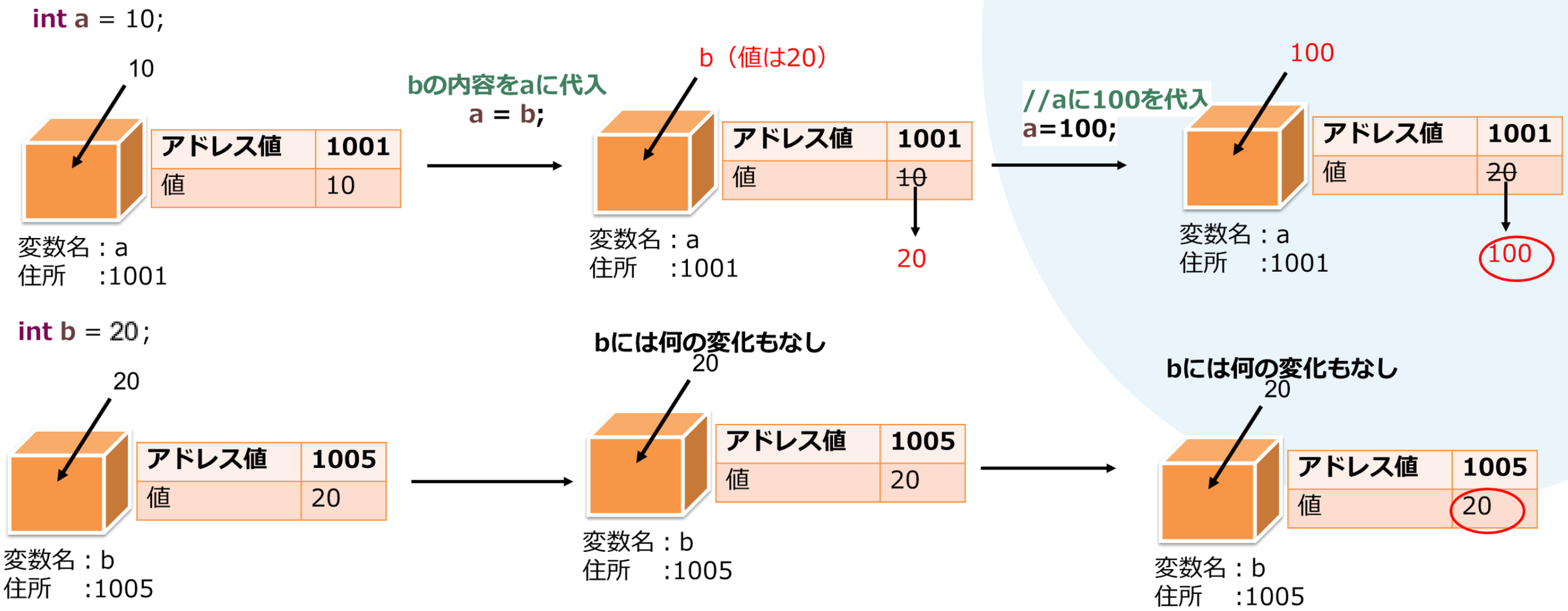
解答

```java
public class Main {

    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        //bの内容をaに代入
        a = b;
        //aに100を代入
        a=100;
        System.out.println("aの値は"+a);
        //bに20を加算
        System.out.println("bの値は"+b);
    }

}
```



- では、どうしてaの値が100、bの値が20になったのかを説明できますか？

基本データ型の値のコピー



- 基本データ型の変数同士で値を代入する場合、値そのものがコピーされ、元の変数とコピー先の変数は独立した存在となります。

- 値の代入後、片方の変数の値を変更しても、もう一方の変数には影響を与えません。

このように、**基本データ型の値の代入操作**は、**メモリアドレスを共有するのではなく、値そのものをコピーすることによって行われます**。これにより、**各変数は独立して存在し、それぞれの値が他方の変数によって影響を受けることはありません**。

Woodland Academy

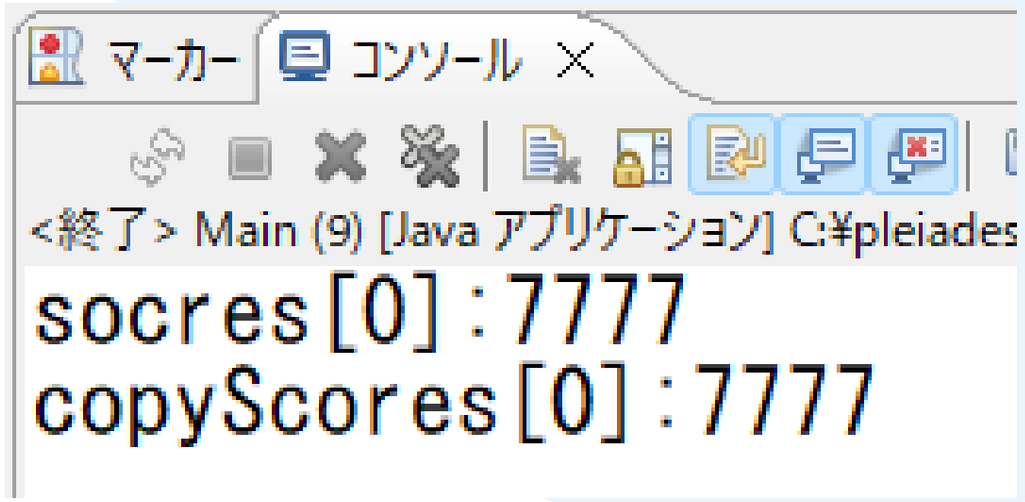
考えてみよう

- scores[0]の値とcopyScores[0]の値は何になるかを考えてみてください

```java
public class Main {

    public static void main(String[] args) {
        //学校のテストの点数を格納する配列を作成
        int[] scores = { 100, 65, 82, 54 };

        /*scores配列の内容をコピーして保持する
         * copyScoresを宣言し、scoresを代入する*/
        int[] copyScores = scores;

        //copyScoresのindex番号0に7777を代入
        copyScores[0] = 7777;

        //scoresのindex番号0の内容を表示
        System.out.println("socres[0]:" + scores[0]);

        //copyScoresのindex番号0の内容を表示
        System.out.println("copyScores[0]:" + copyScores[0]);
    }

}
```

解答

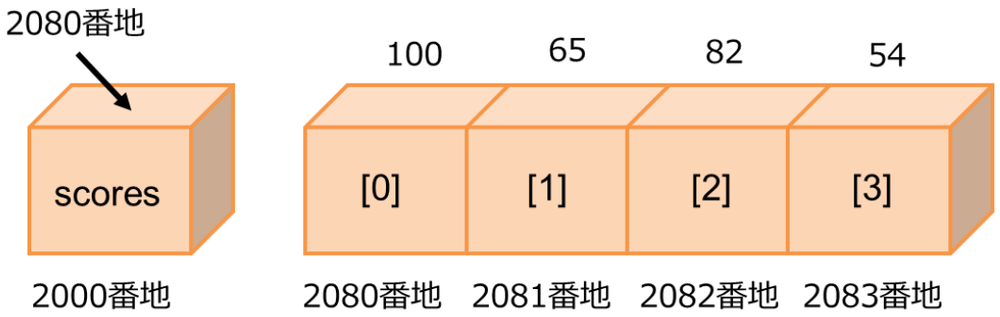
```java
public class Main {

    public static void main(String[] args) {
        //学校のテストの点数を格納する配列を作成
        int[] scores = { 100, 65, 82, 54 };

        /*scores配列の内容をコピーして保持する
         * copyScoresを宣言し、scoresを代入する*/
        int[] copyScores = scores;

        //copyScoresのindex番号0に7777を代入
        copyScores[0] = 7777;

        //scoresのindex番号0の内容を表示
        System.out.println("socres[0]:" + scores[0]);

        //copyScoresのindex番号0の内容を表示
        System.out.println("copyScores[0]:" + copyScores[0]);
    }

}
```



- では、どうしてscores[0]の値とcopyScores[0]の値が7777になったのかを説明できますか？

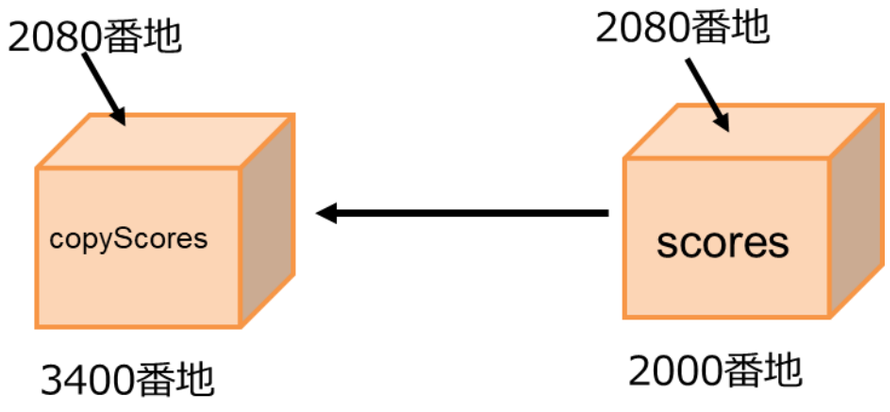
参照型のコピー

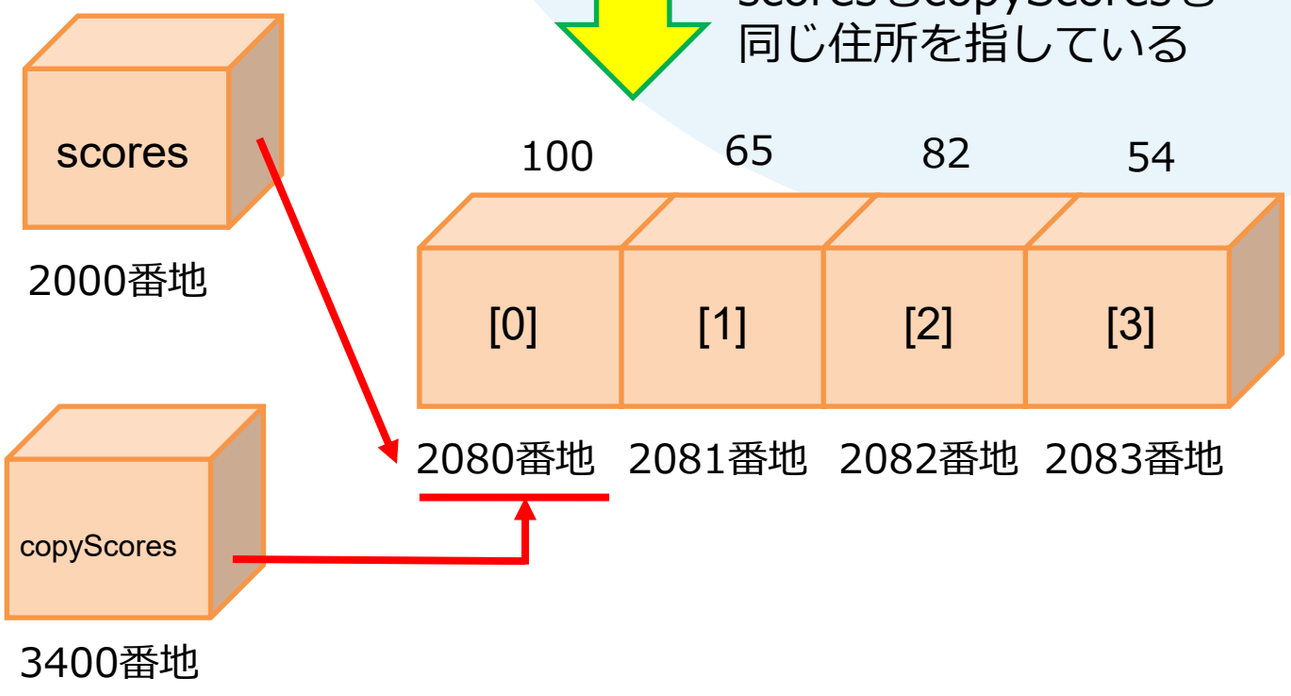
```
int[] scores = { 100, 65, 82, 54 };
```



2000	・・・・	2080	2081	2082	2083
2080		100	65	82	83
3400					
2080					

```
int[] copyScores = scores;
```

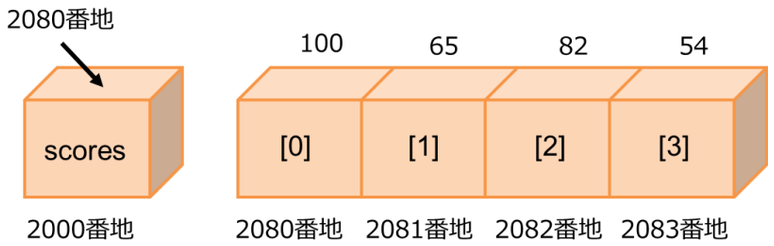




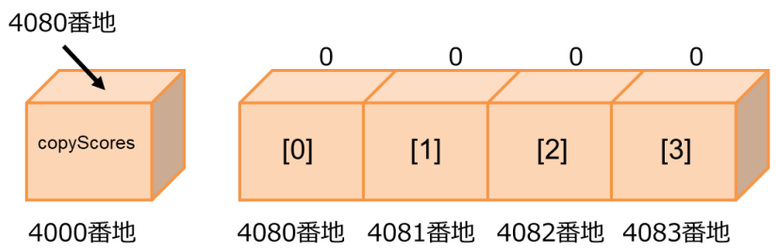
- Javaでは、配列やクラスは参照型として扱われます。参照型の変数を別の変数に代入すると、実際には値そのものではなく、その値が入っている場所の参照（メモリアドレス）がコピーされます。
- そのため、複数の変数が同じ住所を参照することができます。これにより、一方の変数を通じて値をを変更すると、その変更は他方の変数からも反映されます

同じアドレス値を見ないコピー方法

```
int[] scores = { 100, 65, 82, 54 };
```

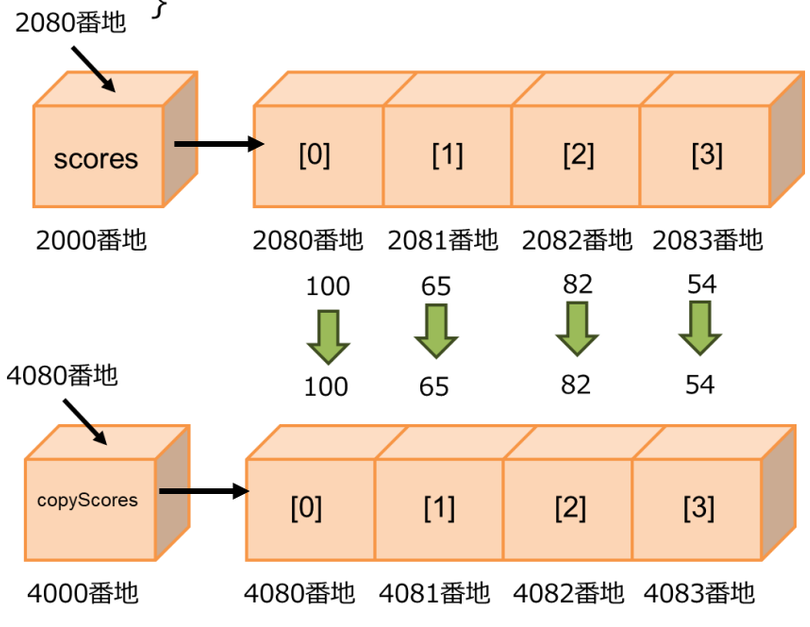



```
int[] copyScores = new int[scores.length];
```

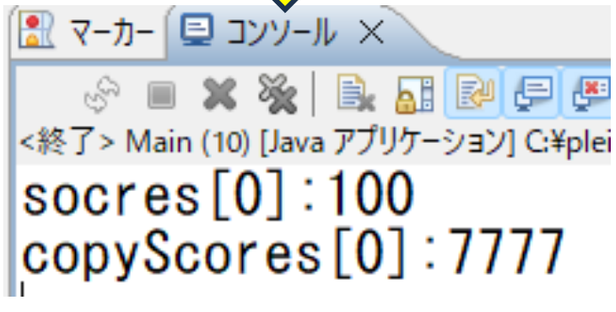


2000	・・・・	2080	2081	2082	2083
2080		100	65	82	83
4000	・・・・	4080	4081	4082	40
4080		0	0	0	0

```
for(int i = 0; i<scores.length; i++) {
    copyScores[i] = scores[i];
}
```




```
public class Main {

    public static void main(String[] args) {
        //学校のテストの点数を格納する配列を作成
        int[] scores = { 100, 65, 82, 54 };

        //「scores.length」の部分は、コピーする要素の数を指定
        int[] copyScores = new int[scores.length];

        //scoresの内容をcopyScoresにコピーする
        for(int i = 0; i<scores.length; i++) {
            copyScores[i] = scores[i];
        }

        //copyScoresのindex番号0に7777を代入
        copyScores[0] = 7777;

        //scoresのindex番号0の内容を表示
        System.out.println("socres[0]:" + scores[0]);

        //copyScoresのindex番号0の内容を表示
        System.out.println("copyScores[0]:" + copyScores[0]);
    }

}
```



コピー元（scores）のインデックス番号を指定して値を取得した後,コピー先の（copyScpres）のインデックス番号を指定して値を代入している

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

```java
Cat a = new Cat("Alice");
Cat b = new Cat("Bob");
a = b;
a.setName("Charlie");
System.out.println("a = " + a.getName()); // => a = Charlie
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**[Deep Copy]）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();
```

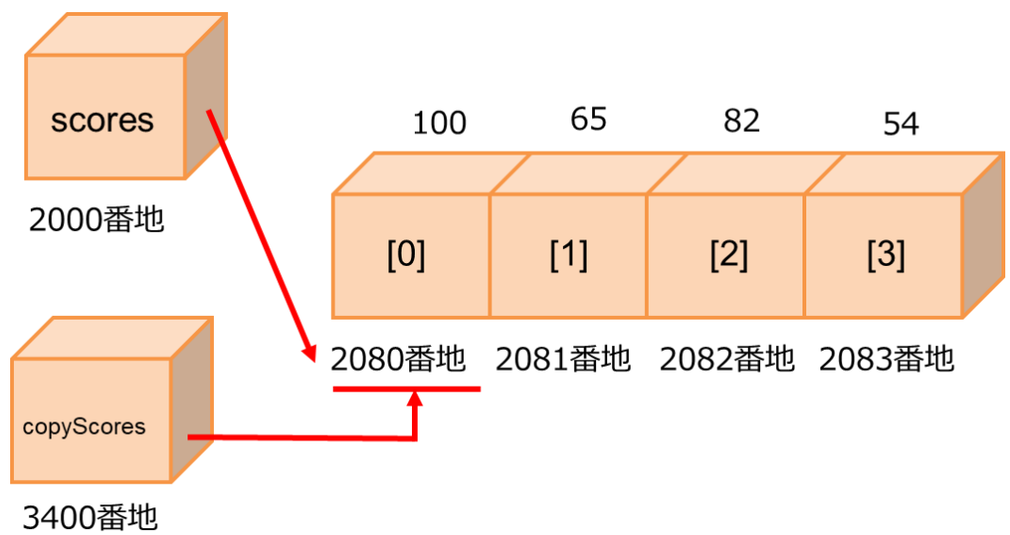
シャローコピーとディープコピー

- シャローコピーとは、コピー元とコピー先が同じデータを参照していること。
- ディープコピーとは、コピー元とコピー先が異なるデータを参照していること。

シャローコピー

コピー元とコピー先が同じ
データを参照していること

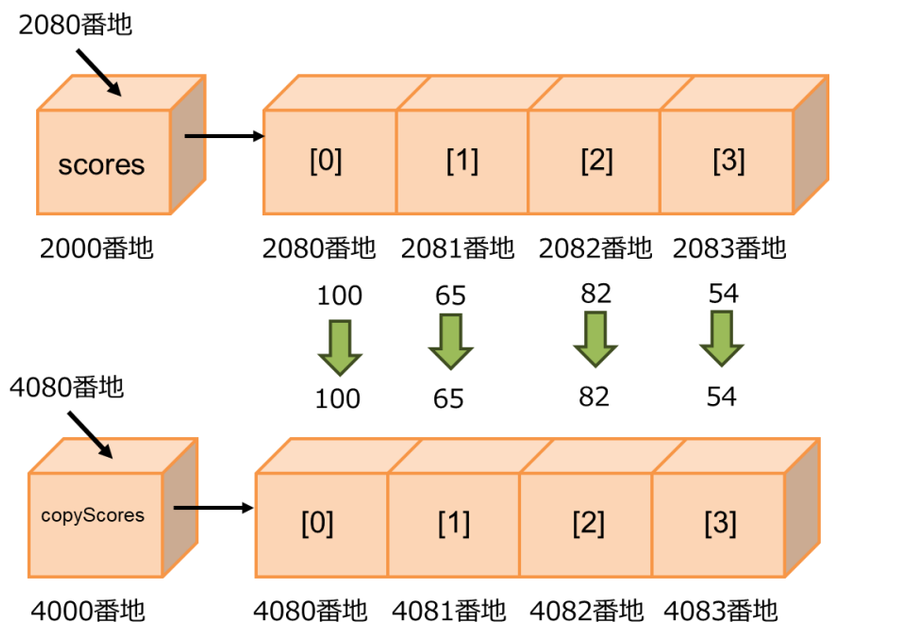
```
int[] scores = { 100, 65, 82, 54 };

int[] copyScores = scores;
```



ディープコピー

コピー元とコピー先が異なる
データを参照していること。

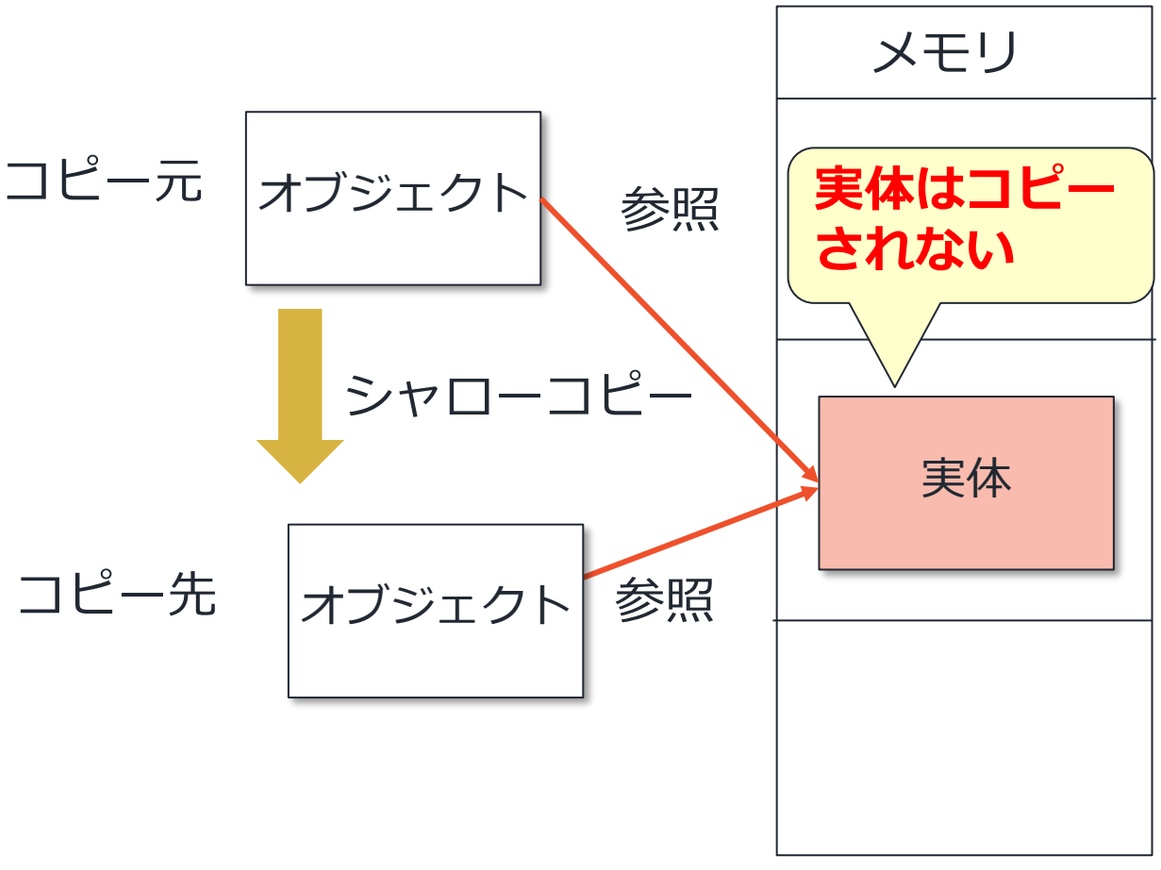
```
int[] scores = { 100, 65, 82, 54 };

int[] copyScores = new int[scores.length];

for(int i = 0; i<scores.length; i++) {
    copyScores[i] = scores[i];
}
```



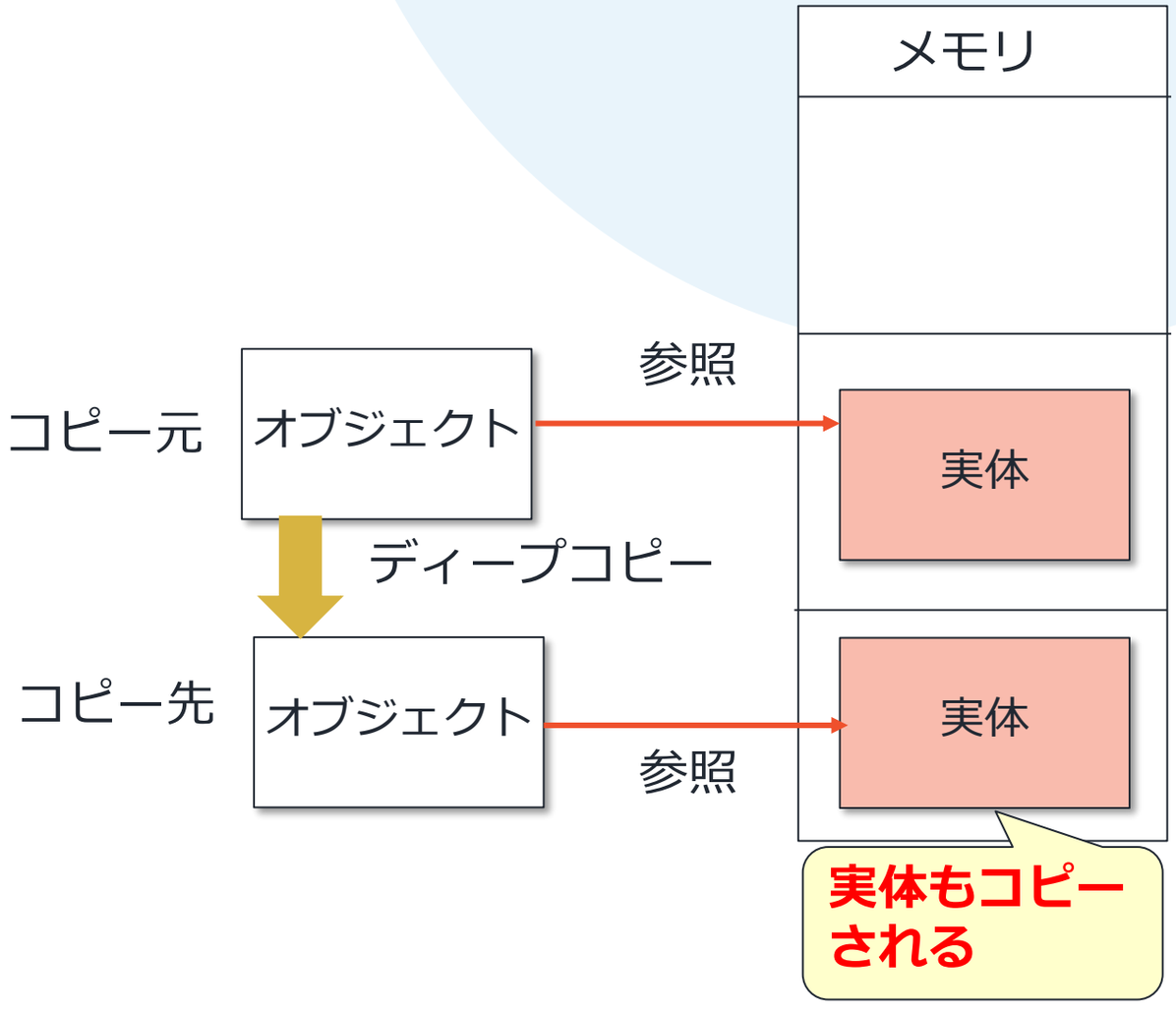
Woodland Academy

シャローコピーとディープコピーまとめ

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

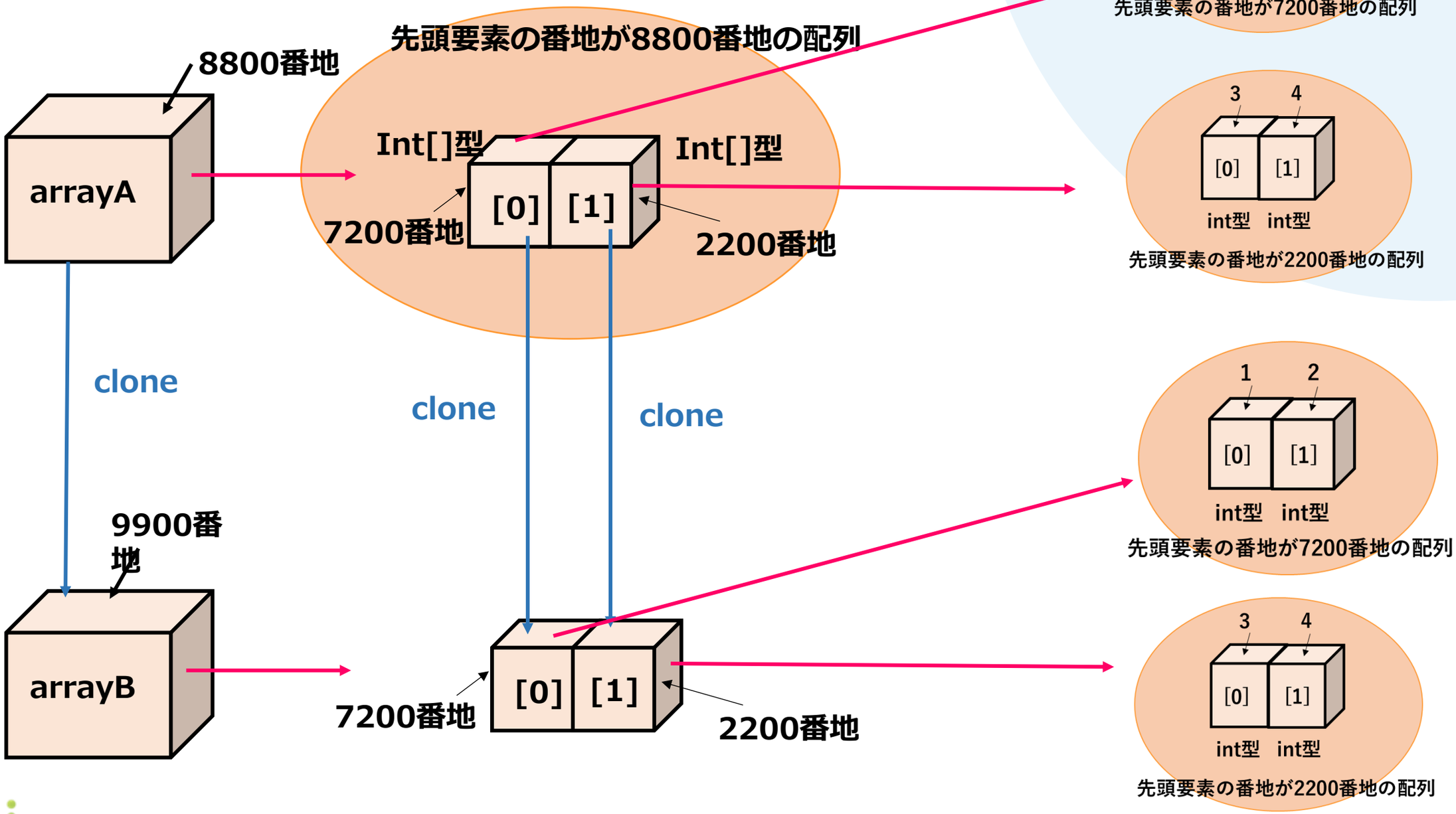
TwoDCopy.java

- 上記のコードを参考にしましょう！

Woodland Academy

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。

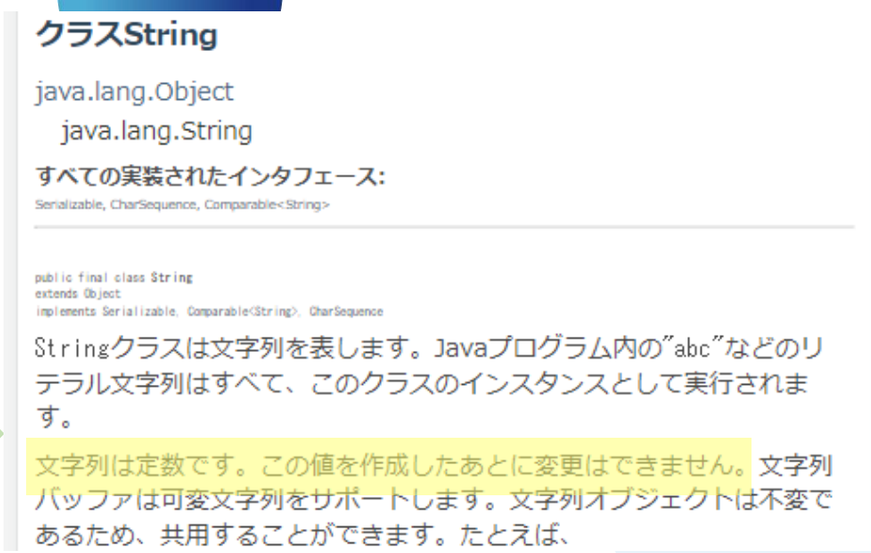
- 実際に比較されたのは、オブジェクトへの**参照**になります：

```
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


クラスString

java.lang.Object
  java.lang.String

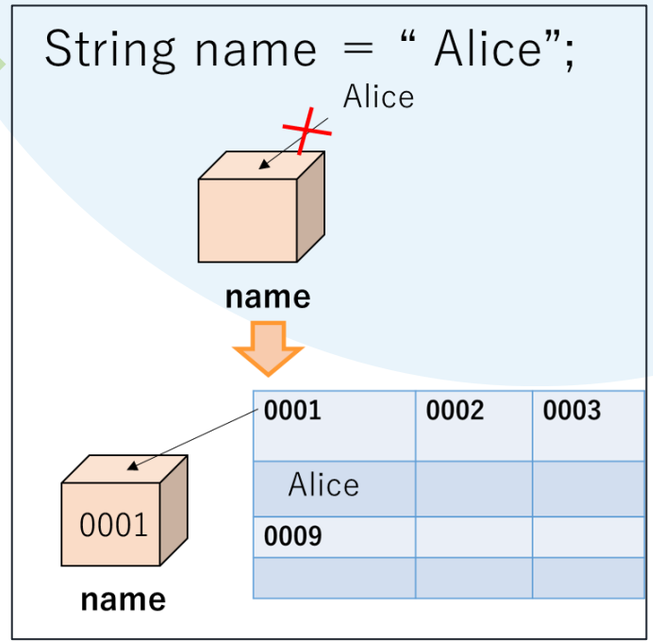
すべての実装されたインタフェース:
Serializable, CharSequence, Comparable<String>

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

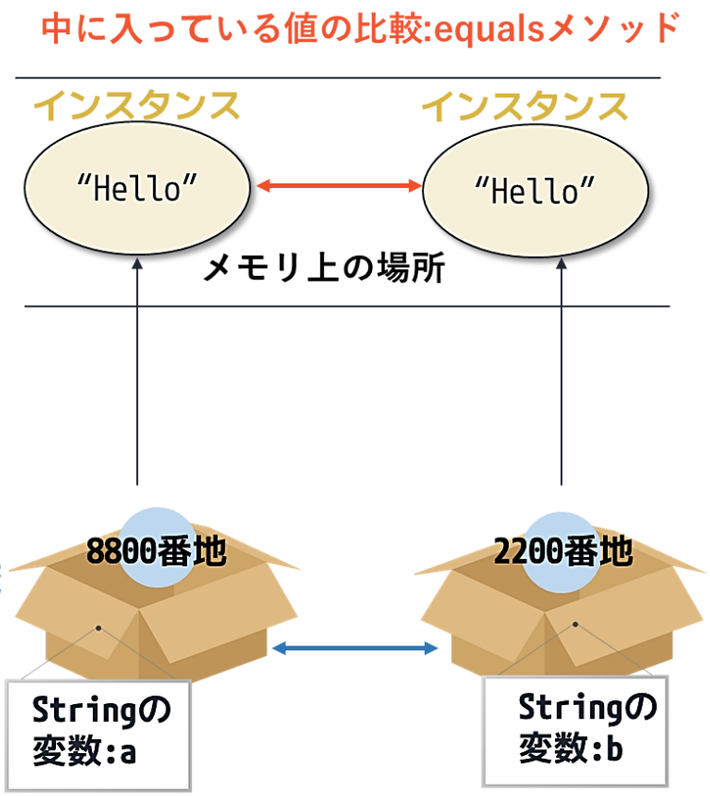
Stringクラスは文字列を表します。Javaプログラム内の"abc"などのリテラル文字列はすべて、このクラスのインスタンスとして実行されます。

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、

- 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。



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!

Woodland Academy

Q&A

まとめ

Sum Up

1.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。

2.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。

Woodland Academy

Thank you!

From Seeds to Woodland — Shape Your Future.

Shape Your Future