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

www.lighthouseit.academy

目次

LightHouse Academy

抽象クラス

- **抽象クラス**[Abstract Class]とは、特殊のクラスの一種で、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**[Abstract Method]）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);
```

LightHouse Academy

・抽象クラスをずばり一言で表現するなら、「処理内容が確定していないクラス（メソッド）」のことです。これが何を意味するかは後述するとして、まずは抽象クラス、抽象メソッドの簡単なサンプルを紹介します。
・スライドのソースを見せてabstractの部分にマーカーや線を引く
・abstract自体は修飾子のひとつです。修飾子とはpublicやprivate、protectedといったアクセス修飾子やstaticやfinalのことです。abstractもこれらと同じく、クラスやメソッドの宣言時に修飾子の一つとして記載します。
クラスやメソッドを定義する時にabstractを加えることで「このクラス（あるいはメソッド）は抽象クラス（メソッド）です」と指定するのです。
書き方については抽象クラスの方はabstractが付いている以外、従来のクラス定義と変わりありません。しかし抽象メソッドの方は見慣れないですね。メソッド名()の後に{}がありません。
抽象メソッドは処理内容が確定していない。つまり、処理内容を書く必要がないということです。必要がなければ、処理を書くためのブロックも必要ありません。そのため、このような記述になります。抽象メソッドを定義する時は{}の代わりに行末に;を書くことを覚えておきましょう。

ソース例1

```java
public abstract class Animal {
    // メンバ変数
    protected String name;

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

    // 鳴く
    public void sounds() {
        System.out.println(name + " make sounds");
    }

    // 未実装（何も処理を書かないメソッド）
    protected abstract void eat(String food);

}
```

```java
public class Main {

    public static void main(String[] args) {
        // Animalクラスをインスタンス化するとエラー
        // Animal animal = new Animal("animal");
        // ⇒Cannot instantiate the type Animal

        Cat kitty = new Cat("kitty");
        kitty.eat("cat food");
        // ⇒kitty ate cat food

    }

}
```

```java
public class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void eat(String food) {
        System.out.println(name + " ate " + food);
    }

}
```


抽象クラスをインスタンス化しようとしても、コンパイルできずエラーとなります。
ルールに則って作ったクラスなのにエラーが出るなんていじわるじゃないか！と思われるかもしれません。しかし、ここは一つ落ち着いて考えてみてください。抽象クラスは「処理内容が確定していないクラス」でしたね。つまり未完成な設計図です。
処理内容が決まっていないクラスがインスタンス化できてしまうということは、未完成な設計図からモノが作られてしまうということです。まともに動かないものが作られてしまうのはとてもマズイことです。抽象クラスがインスタンス化できないのはいじわるではなく、むしろJavaの優しさなのです。
では未完成な設計図をどう扱えばいいのか。継承してサブクラスで具体的な処理を実装して完成させればいいのです。そしてこの完成した設計図こそが、具象クラスです。処理内容が未確定というあいまいな状態（抽象的）から、処理内容が確定した状態（具体的）になったのです。
ちなみに、サブクラスもabstractで抽象クラスに指定すれば、抽象メソッドをオーバーライドしていなくてもエラーにはなりません。サブクラスにオーバーライドが強制されるのは、具象クラスのときだけです
（ソース例2）を見せる
☆Point!☆
- 抽象クラスはインスタンス化できないことで、未完成のインスタンスが生まれることを防いでくれる
- すなわち、抽象クラスは継承元（スーパークラス）となることが前提のクラス
- 抽象クラスの継承先（サブクラス）をふたたび抽象クラスにすることもできる

インタフェースとの使い分け
抽象クラス・・・**クラスごとに共通する**部分があったとき

インターフェース・・・**機能を足したい**とき。

ソース例2

```java
public abstract class LivingThing {
    // 未実装（何も処理を書かないメソッド）
    protected abstract void sleep();

}
```

```java
public abstract class Animal extends LivingThing {
    // メンバ変数
    protected String name;

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

    // 鳴く
    public void sounds() {
        System.out.println(name + " make sounds");
    }

    // 未実装（何も処理を書かないメソッド）
    protected abstract void eat(String food);

}
```

```java
public class Cat extends Animal{
    public Cat(String name) {
        super(name);
    }

    @Override
    public void eat(String food) {
        System.out.println(name+" ate "+food );
    }

    @Override
    protected void sleep() {
        System.out.println(name+" sleep ");

    }
}
```

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：

```java
public class Cat extends Animal {
    @Override
    public void eat(String food) {
        System.out.print(getName() + " ate " + food + ", ");
        meow();
    }
}
```

- 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げる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

LightHouse Academy

復習として読み上げるだけでよい
既に解説済みであるため。

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ー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**。

LightHouse Academy

インタフェースと継承の部分で話をしているため、復習としてスライドを読み上げる
抽象クラスはインタフェースと違って多重継承することはないところは重要なので、マーカーなどの線を必ず引くようにする。

LightHouse Academy

目次

LightHouse Academy

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 }
```

LightHouse Academy

thisキーワードについては、すでに話しているはずなので、
スライドを読み上げるだけでよい

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。

LightHouse Academy

ここからは新規の内容になるので、superを含めて話をする

ソース例

```
public class Person {

        String name;

        Person(){
                this("未設定");
        }

        Person(String name) {

                this.name = name;
        }
}
```

```
public class Student extends Person {

        int stuNo;

        Student(String name, int stuNo){
                super(name);
                this.stuNo = stuNo;
        }

}
```

LightHouse Academy

this(引数)、super(引数)です。
このキーワードにすぐ（）を書いてあげることで、コンストラクタを呼び出すことができます。
クラスの中でいくつかコンストラクタを書きたい。そしてコンストラクタの中で、似たような処理をさせたい。
でも、同じ処理を何回も書くのは嫌だから、このthisとsuperを使ってそれぞれ呼び出して利用しようというための記述です。
This()は同じクラスの中でコンストラクタを呼び出しあうことができます。
Super（）のほうは、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。
実際の例をプログラムで見ていきます。
クラスPersonから見ていくと、さっきと同じようにメンバ変数nameを持っていて
コンストラクタが２つ定義されています。
引数なしのPersonそれから、String nameを受けとるPersonがある
引数nameを受け取るコンストラクタは、このnameをthis.nameに代入してあげる
引数なしのコンストラクタ　Personは　名前がわからないので、とりあえず、未設定という文字列を設定しておく。
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Person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。

そうするとどうなるかというと、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Personに置き変わるイメージを持ってください。

そうするとこのように、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。

今度は、サブクラスのstudentの方ですね。
コンストラクタ引数2つのものを用意しています。
これは、名前と学籍番号を設定するコンストラクタになるのですけれども、名前の方は、スーパークラスのコンストラクタの方で設定してありますので、これを呼び出して利用しようというのが、Super（name）になります。これは、スーパークラスのコンストラクタつまり、引数1つのコンストラクタを呼び出すので、このPerson（String name）を実行することになります。
そして、stuNoは自分のところで代入しようねという処理になっています。
このように、this()やsuper()を利用して処理を簡単に書けるようになります。
前にもやりましたが、super（）はなければ自動的に挿入されることは、やりましたよね。
なぜ、先頭に書かなきゃいけないも復習ですが、話しておくと、
スーパークラスから順々に変数などを作っていく流れになる。だから先頭に書くというお話をしましたよね。

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**同名メソッド**（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。

最後にそのまま読み上げて終了


Q&A
LightHouse Academy
© Suporich Co.Ltd.
15

目次

LightHouse Academy

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。

LightHouse Academy

ここからは、井澤の場合は、自身で作成した補足資料を使用して説明をしている
よろしいければお使いください

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

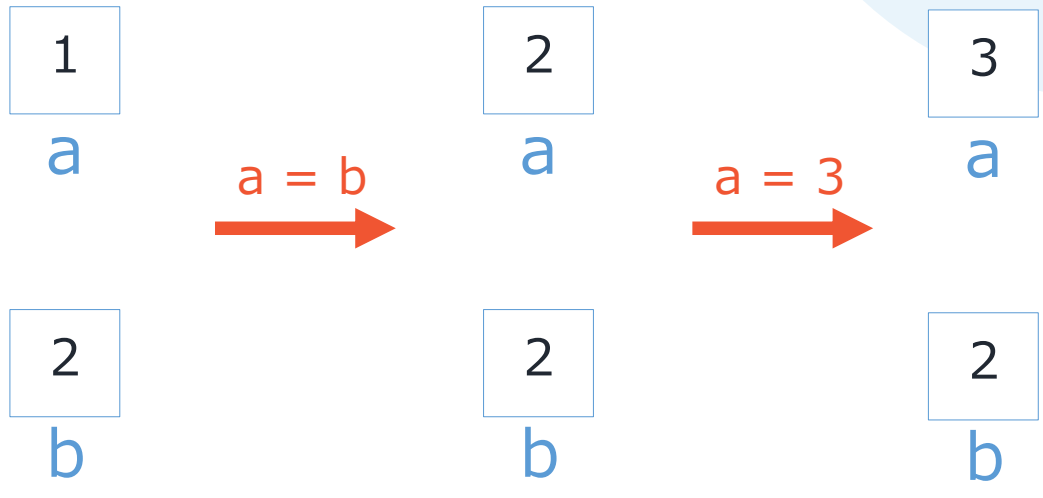
Example

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]);
```

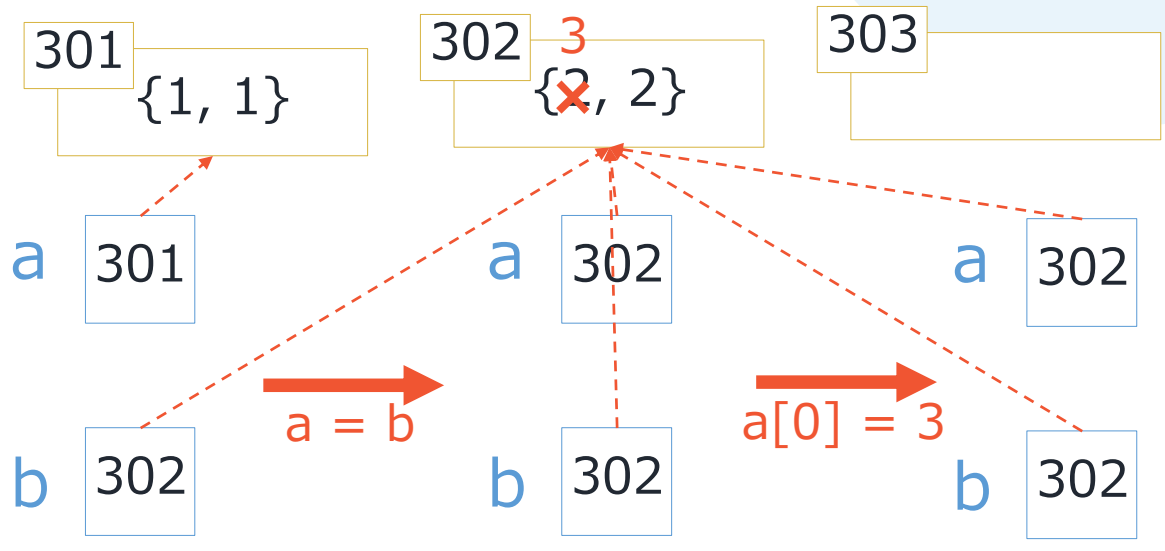
基本データ型を格納する方法

- 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**[Reference]が格納されるます。 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**[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：

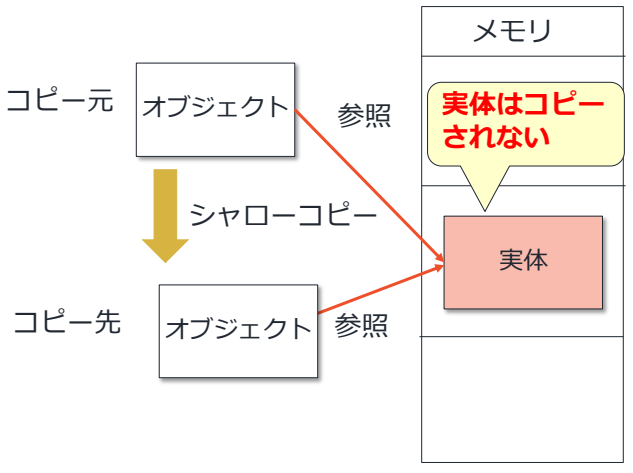
```
1 int a = {1, 1};
2 int b = {2, 2};
3 a = new int[2];
4 for (int i = 0; i < 2; i++) {
5     a[i] = b[i];
6 }
7 a[0] = 3;
8 System.out.println("a = " + a[0] + " " + a[1]); // => a = 3 2
9 System.out.println("b = " + b[0] + " " + b[1]); // => b = 2 2
```

- 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：

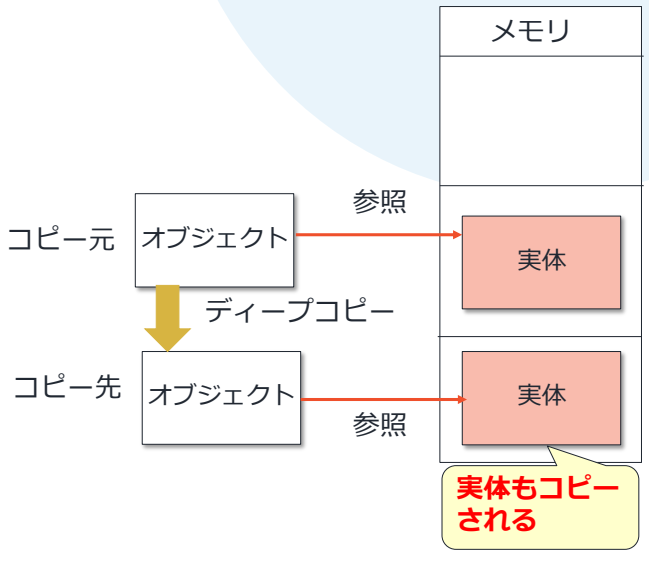
```
a = b.clone();
```

シャローコピーとディープコピー

シャローコピー



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？

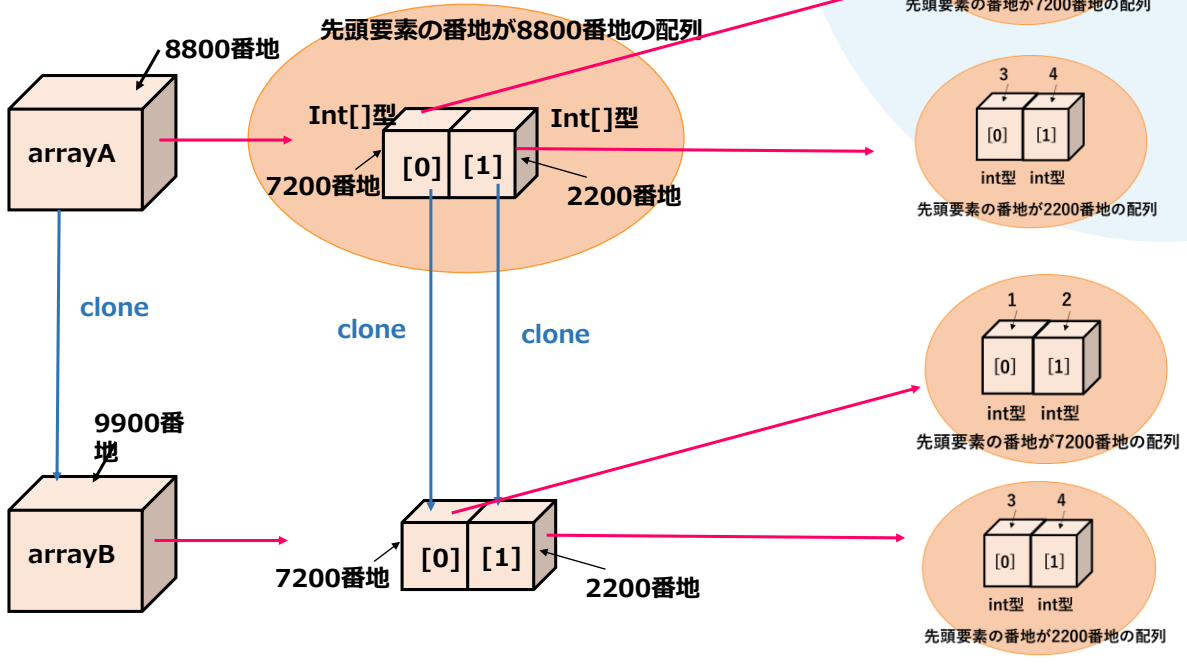
Try 01011 11010 01011

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
```

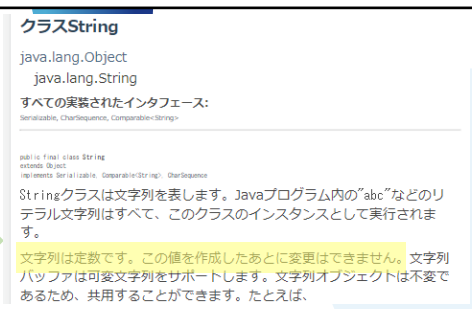
- この例では、a と b は**異なる**配列ですから、たまたま同じ値を含んでも、「a == b 」は「false」を返します。

LightHouse Academy

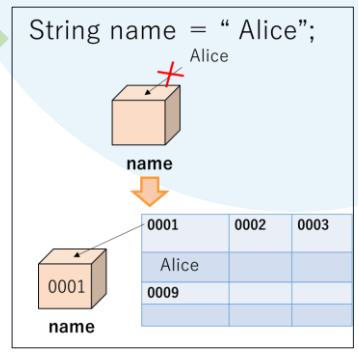
文字列の比較について
このままスライドを読み上げるだけでよい

Stringの特徴

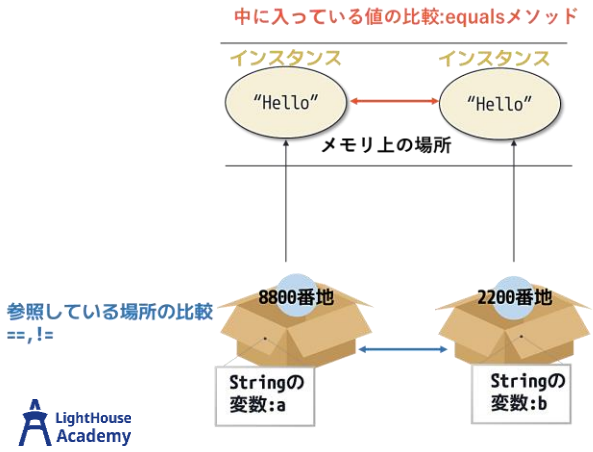
- Stringは、定数である
  ・つまり変更できない



- 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。




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

LightHouse Academy

そのままよみあげるだけでよい

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!

目次

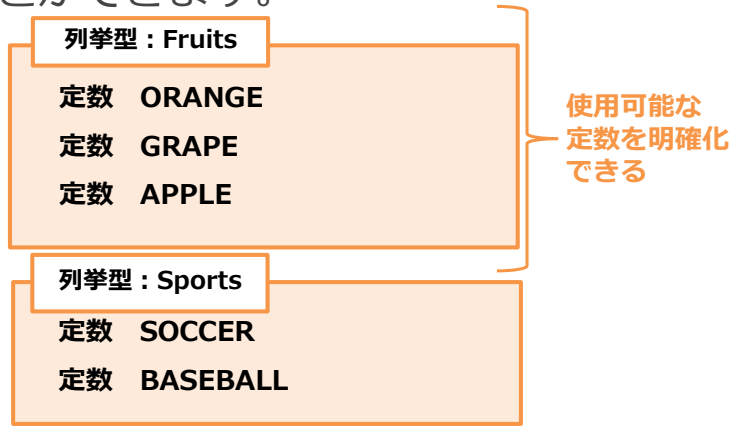
LightHouse Academy

列挙型

- **列挙型**[Enum]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。

LightHouse Academy

これ以降の説明については、スライドの内容を見ても生徒が理解できないため、
以下のリンクを画面共有してそのまま読み上げていく
https://camp.trainocate.co.jp/magazine/java-enum/

ソースについては、ソース例を参照

ソース例

```java
public enum Fruit {
    //定数を書く
    APPLE,ORANGE,CHERRY;

    public static void main(String[] arg) {
        switch(Fruit.APPLE) {
        case ORANGE:
            System.out.println(ORANGE +"is yummy");
        break;
        case APPLE:
            System.out.println(APPLE +"is yummy");
        break;
        case CHERRY:
            System.out.println(CHERRY +"is yummy");
        break;
        }


    }

}
```

ソース例2

```java
public enum Fruit {
    ORANGE("オレンジ"),APPLE("りんご"),CHERRY("さくらんぼ");

    //定数の説明（和名）を格納するためのメンバ変数
    private String japanese;

    private Fruit(String japanese) {
        this.japanese = japanese;
    }

    //和名を表示する
    public String getJapanese() {
        return japanese;
    }

}

```

```java
public class Main {

    public static void main(String[] args) {
        // APPLEの和名はりんご
        System.out.println(Fruit.APPLE + "の和名は" + Fruit.APPLE.getJapanese());

    }

}
```

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 型の変数が取り得る値は、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。

```java
public enum State {
    RUNNING(1.0f), POWERED_DOWN(0.0f), SLEEPING(0.2f); // ここの「;」は省略不可

    private float power;

    State(float power) {
        this.power = power;
    }

    public float getPower() {
        return power;
    }
}
```

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。

目次

LightHouse Academy

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、ダイヤのクラスは、車のクラスだけに使われています。この場合、ダイヤクラスを車クラスの内部クラスとして書いてもいいでしょう：

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

LightHouse Academy

クラスのメンバーはと聞かれて思いつくのは、フィールド変数とメソッドです。その他にクラスもあるよ、と言われると「？？」となりますよね。クラスの中にクラスを定義する、これを内部クラス（インナークラス）といいます。

内部クラスはクラスをメンバーと同じように扱うことができます。メンバーと同じようにprivate、protected、publicといったアクセスレベルを付与することができます。また、内部クラスからは同じクラス内のフィールド変数、メソッドを参照することができます。

ですので、あるクラス内のフィールド変数やメソッドにアクセスするクラスを宣言する場合に内部クラスとして宣言します。あるクラスのフィールド変数やメソッドに別のクラスからアクセスする方法としては、継承してサブクラスを宣言する方法があります。

でも、そこまでしたくないというときに内部クラスを宣言すると便利です。内部クラスはメソッド内でも宣言することもできます。その場合、メソッド内のみで機能します。

なお、内部クラスを持つクラスのことを外部クラスといいます。

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。

内部クラスには、staticかstaticでないかで2種類の書き方が存在する
次のページでソース例を紹介する
ここは、生徒のにソースを書かせるなどはしなくてよい

ソース例　非静的

```java
public class Sample1 {
    String str1 = "外部クラスのフィールドです。";

    public void outerExe() {
        System.out.println("外部クラスのメソッドを実行しました。");
        //innerExe();//外部から内部を読んだらエラーになる。
    }

    class Sample2 {
        String str2 = "内部クラスのフィールドです。";
        public void innerExe() {
            System.out.println("内部クラスのメソッドを実行しました。");
            //外部クラスのフィールドにアクセス
            System.out.println(str1);
            //外部クラスのメソッドにアクセス
            outerExe();
        }
    }

}
```

LightHouse Academy

内部は外部が見えるが、外部は内部が見えない

そもそも、内部クラスが作成されるかどうかは、開発者次第となるため、絶対に内部クラスがあるとは限らない
そのため、あるかどうかもわからないものは呼び出せないということで、外部クラスから内部クラスは呼び出せない

だが、内部クラスは、自分の上を見上げれば何が書いてあるかを確認することはもちろんできるため、外部クラスの内容を問題なく呼び出すことができる

ソース例　静的

```java
//外部クラス
class OuterClass {
    // staticなフィールド
    private static String str = "OuterClassの変数";
    private String name ="akemi";
    public void outerEx() {
        this.name =str;
    }

    // staticな内部クラス
    static class InnerClass {
        String str2 = "内部クラスのフィールドです。";
        public void innerSample() {
            //外部クラスのフィールドにアクセス
            System.out.println(str);
            //外部クラスのメソッドにアクセス
            //outerEx();//エラー
        }
        public void innerSample2() {
            //System.out.println(name);//エラー
            System.out.println("内部クラスのメソッドです。");
        }
    }

```


staticについては、オブジェクト指向でやった考えと一緒である
staticからstaticは呼び出せるか、staticから非staticは呼び出せない

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、異なるクラスを一緒に**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

前へ

- 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：

```java
public class Car {
    private Driver Driver;
    private String brand;

    private class Driver {
        public String getCarBrand() {
            return brand;
        }
    }
}
```

Try Car.java

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
LightHouse Academy
© Suporich Co.Ltd.
50

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!