

オブジェクト指向 2

クラスとインスタンス

2017/10/30(月) プログラミングII 第四回
福井大学 工学研究科 情報・メディア工学専攻
長谷川達人

※WindowsでPC起動しておいてください。



本講義の概要

前半：Java 担当：長谷川		後半：Scala 担当：石井先生	
第1回	Java基礎文法/開発環境の使い方	第9回	Scala言語の基本
第2回	Java基礎文法/C言語との差異	第10回	関数型プログラミングの基本
第3回	オブジェクト指向	第11回	関数とクロージャ
第4回	クラス/インスタンス/メソッド1	第12回	関数型プログラミングの簡単な例
第5回	クラス/インスタンス/メソッド2	第13回	静的型付けと動的型付け
第6回	継承/カプセル化1	第14回	ケースクラス/パターンマッチング
第7回	継承/カプセル化2	第15回	多相性やパターンマッチングの例
第8回	中間試験/Java言語のまとめ	第16回	期末試験

※来年以降は全てJavaになる予定

本日の目標

概要

Javaで最も重要である**オブジェクト指向**の第一歩として
「ファイルを分割するため」だけではない
クラスの使い方について学ぶ.

目標

クラスとインスタンスの違いがわかる
クラスとインスタンスの使い方がわかる



なるほど

本日の提出課題

課題を意識しながら授業を聞くとよい

課題 1

本日の授業を聞いて、
初めて知ったと思う内容を2点簡潔に述べよ。
「簡潔に述べよ」≡「1, 2文程度で述べよ」と考えるとよい

課題 2

本日の授業を聞いて、
質問事項または**気になった点**を2点簡潔に述べよ。
質問を考えるときは、様々な点に対して5W1Hを考えるとよい

本日の目次

- 復習
- クラスとインスタンス 1
- オブジェクト指向プログラミング
- クラスとインスタンス 2
- Javaのメモリ領域
- 命名規則
- 修飾子



多いけど時間内に
終わるのか．．．
いや，終わらない．

メソッドのまとめ

- 使用方法は基本的にはC言語と同じである.

```
[メソッド名]([引数], ...); // 呼び出し方法  
public static [戻り値] [メソッド名]([引数], ...){ // 宣言方法
```

- プロトタイプ宣言は不要である.
- 変数のスコープ（有効範囲）はブロック単位（{中括弧}で囲まれた範囲内）である.
- 引数と戻り値には配列を用いることができる.
- 通常型は値渡しに対し配列等参照型は参照渡しである.
- 引数の数か型を変えれば、同じ名称のメソッドを多重定義することができる.

クラスを学ぶ前準備

ファイルの分割

```
class Sample1{  
    public static void main(String[] args){  
        Calc.methodA();  
        Calc.methodB();  
        InOut.methodE();  
        InOut.methodF();  
    }  
}
```

呼び出し時は
[クラス名].[メソッド名]();

```
class Calc{  
    メソッドA  
    メソッドB  
    メソッドC  
}
```

```
class InOut{  
    メソッドD  
    メソッドE  
    メソッドF  
}
```

可読性の向上

手間の削減

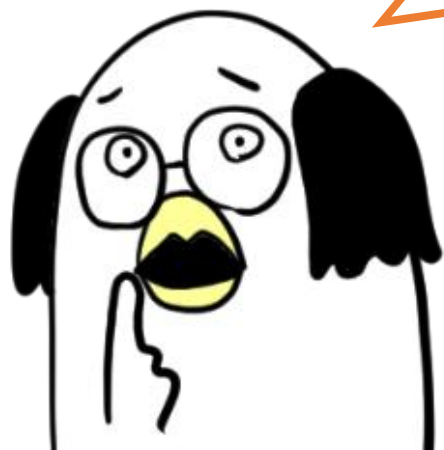
影響範囲の限定

分業の容易さ
クラスファイルごとに
分業がしやすい

役割ごとにクラス
ファイルを分ける。

本日の目次

- 復習
- クラスとインスタンス 1
- オブジェクト指向プログラミング
- クラスとインスタンス 2
- Javaのメモリ領域
- 命名規則
- 修飾子



今日も穴埋めがあります。

クラスとインスタンス

クラスとは、C言語でいう構造体に近いものである。
クラスには、**フィールド**（データ）と**メソッド**（処理）を定義することができる。

Monsterクラス

int hp;

String name;

int attack();



```
public class Monster{  
    int hp = 100;  
    String name = "";  
  
    public int attack(){  
        // 0~9までの乱数のダメージを返す  
        return (int)(Math.random()*10);  
    }  
}
```

クラスとインスタンス

クラスは設計図であり，設計図から生成した実体を
インスタンスという。

Monsterクラス

インスタンス 1
(スライミ)



インスタンス 2
(キメラン)



インスタンス 3
(ドラゴヌ)



どのクラスにもmainメソッドを書くことはできるがややこしいので，今後はMainクラスに各運用とする。

```
public class Main{  
    public static void main(String[] args){  
        Monster m1 = new Monster();  
        Monster m2 = new Monster();  
        Monster m3 = new Monster();  
    }  
}
```

インスタンスの作成
[クラス名] [変数名] = **new** [クラス名]();

クラスとインスタンス

イメージとしては、

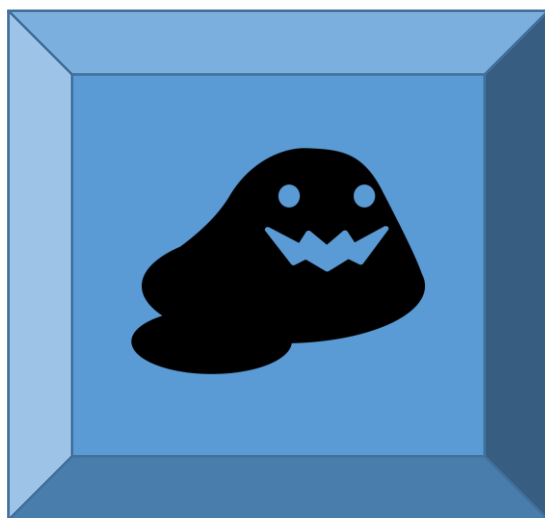
クラス

インスタンス

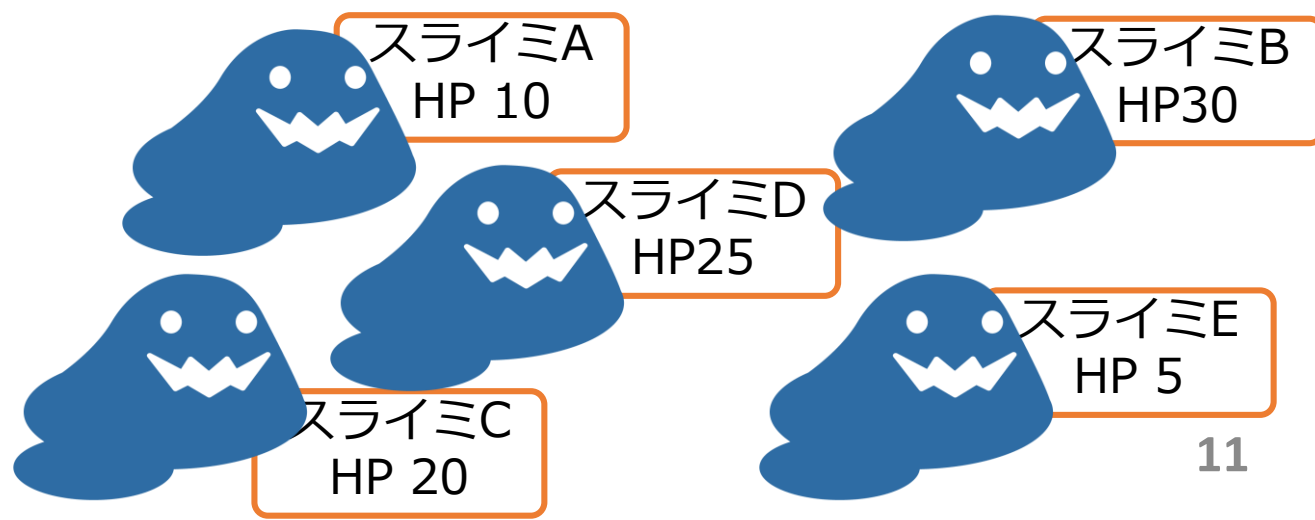
= 設計図や金型

= クラスをもとに作られた量産品

クラス（金型）



インスタンス（金型で量産）



クラスとインスタンス

各インスタンスのフィールドとメソッドにアクセスするには, 「. (ドット)」 で接続する.

```
public static void main(String[] args){  
    Monster m1 = new Monster(); // インスタンスの生成  
    m1.name = "スライミ"; // m1 の名前を設定  
    m1.hp = 100; // m1 のHPを設定  
    System.out.println(m1.name + "の攻撃");  
    System.out.println(m1.attack() + "のダメージ!");  
}
```

m1の名前
を変更

スライミの攻撃
5のダメージ!

m1.attack()
を呼び出し

クラスとインスタンス

各フィールドはインスタンスごとに保持される.

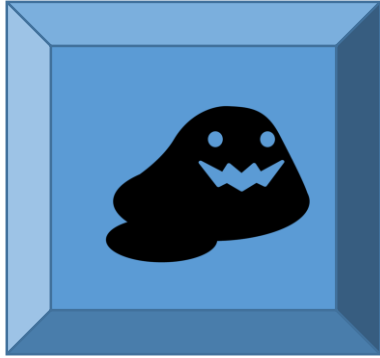
```
public static void main(String[] args){  
    Monster m1 = new Monster();  
    Monster m2 = new Monster();  
    m1.name = "スライミ"; // m1 の名前を設定  
    m2.name = "キメラン"; // m2 の名前を設定  
    System.out.println(m1.name + "が現れた！");  
    System.out.println(m2.name + "が現れた！");  
}
```

m1とm2それぞれに
nameを設定

スライミが現れた！
キメランが現れた！

クラスとインスタンス

金型



Monster
クラスの定義

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    public int attack(){  
        // 0~9までの乱数のダメージを返す  
        return (int)(Math.random()*10);  
    }  
}
```

```
public static void main(String[] args){  
    Monster m1 = new Monster();  
    m1.name = "スライミA"; // m1の名前を設定  
    m1.hp = 100;           // m1のHPを設定  
    Monster m2 = new Monster();  
    m2.name = "スライミB"; // m2の名前を設定  
    m2.hp = 150;           // m2のHPを設定  
}
```

インスタンスの生成



クラスとインスタンス

練習問題 1 : Monsterクラスの実装

P9に記述したMonsterクラスを実装せよ.

また, P13のMainクラスとmain()メソッドも実装し, インスタンスの持つフィールドが独立であることを動作確認せよ. 3体目のモンスターを作成してもよい.

```
public class Monster{
    int hp = 100;
    String name = "";
    public int attack(){
        // 0~9までの乱数のダメージを返す
        return (int)(Math.random()*10);
    }
}
```

```
public class Main{
    public static void main(String[] args){
        Monster m1 = new Monster();
        Monster m2 = new Monster();
        m1.name = "スライミ"; // m1の名前を設定
        m2.name = "キメラン"; // m2の名前を設定
        System.out.println(m1.name + "が現れた!");
        System.out.println(m2.name + "が現れた!");
    }
}
```

クラスとインスタンス

練習問題 2 : Monsterクラスの拡張

このままだと、各モンスターは同じ程度の攻撃しかしてこなくてつまらない。Monsterクラスに、int型フィールド "ap" を定義し、attack()の戻り値はap+乱数とせよ。

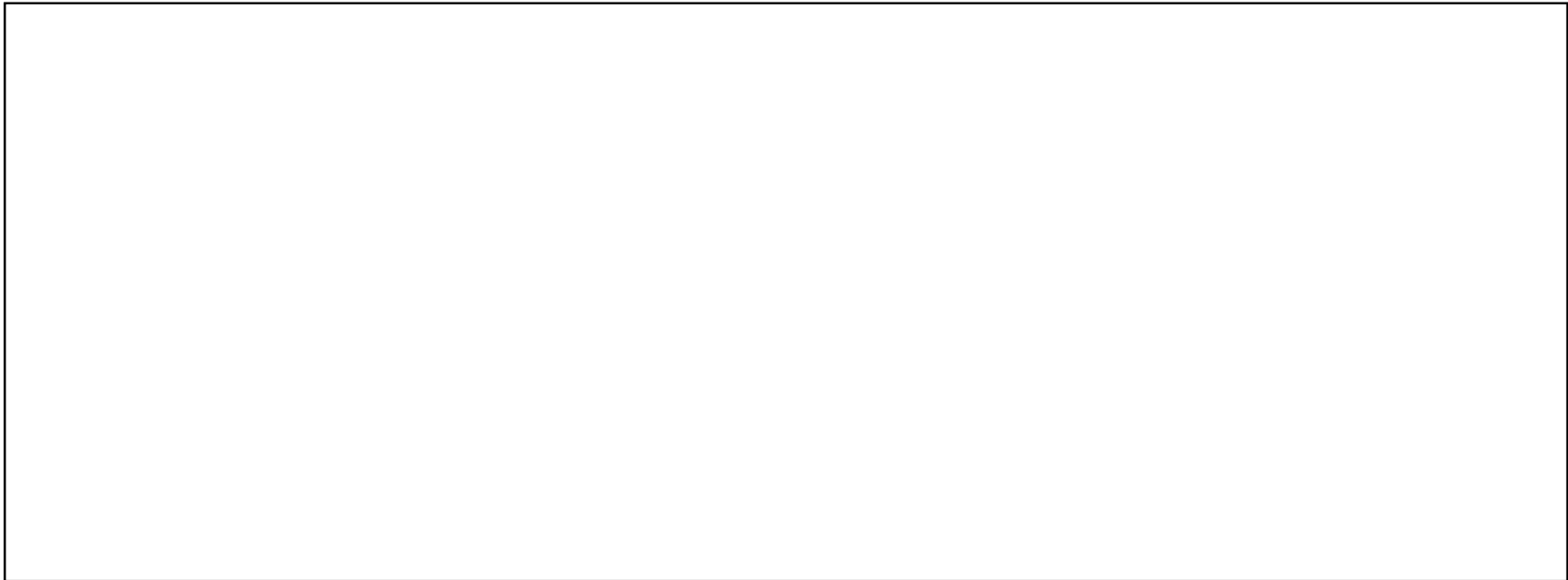
```
public class Main{  
    public static void main(String[] args){  
        Monster m1 = new Monster();  
        Monster m2 = new Monster();  
        m1.name = "スライミ"; // m1の名前を設定  
        m1.ap = 1; // m1の最低攻撃力apを設定  
        m2.name = "キメラン"; // m2の名前を設定  
        m2.ap = 50; // m2の最低攻撃力apの設定  
        System.out.println(m1.name + "の攻撃:" + m1.attack() + "ダメージ");  
        System.out.println(m2.name + "の攻撃:" + m2.attack() + "ダメージ");  
    }  
}
```

動作確認用
main()メソッド

クラスとインスタンス

練習問題 2 : Monsterクラスの拡張

このままだと、各モンスターは同じ程度の攻撃しかしてこなくてつまらない。Monsterクラスに、int型フィールド "ap" を定義し、attack()の戻り値はap+乱数とせよ。



本日の目次

- 復習
- クラスとインスタンス 1
- **オブジェクト指向プログラミング**
- クラスとインスタンス 2
- Javaのメモリ領域
- 命名規則
- 修飾子

オブジェクト指向プログラミング

オブジェクト指向プログラミングとは

現実世界のモノ等の**オブジェクト単位でプログラムを開発していく考え方** のことである。

→諸説あるが、本講義ではこのように定義しておく。

→なお、オブジェクト指向とは何か？という質問に正確に回答することは中々難しい。

→したがって、今は「**現実世界のモノ単位でクラスを分けて開発していくんだなあ**」程度の認識で良い。

オブジェクト指向プログラミング

Java = オブジェクト指向？

Javaの特徴としてオブジェクト指向であると第一回の講義で説明したが、「Java=オブジェクト指向」ではない。

いくらJavaで開発していようと、main()メソッドに順番にプログラムを書き進めていけば、C言語のような手続き型のプログラミングができる。

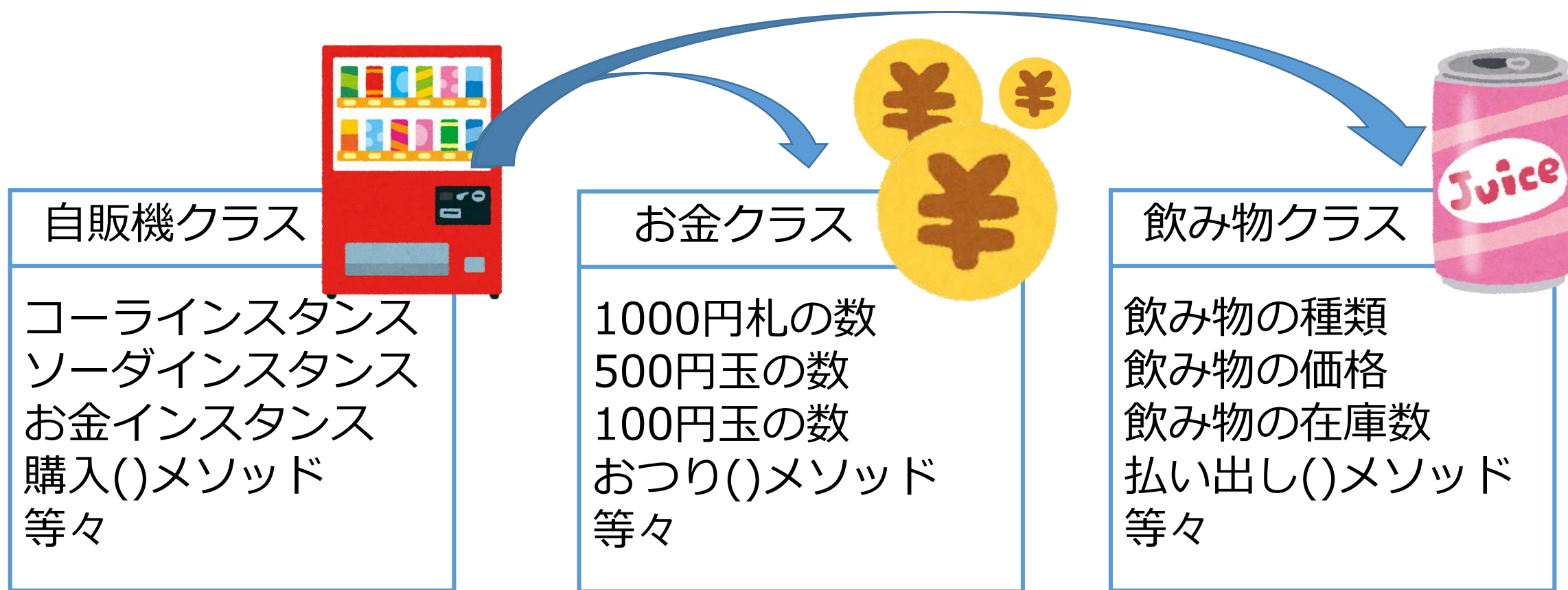
従って、Javaが書ける = オブジェクト指向ができる、
というわけではなく、それぞれに練習が必要である。



オブジェクト指向プログラミング

開発例のイメージ

自動販売機の内部プログラムを考えてみる



オブジェクト指向プログラミング

開発例のイメージ

C言語では、非常に長いメイン関数と多種の関数で書かれるであろう自動販売機プログラムが、Javaではオブジェクトごとに開発される。

どのようなオブジェクトに分割し、各オブジェクトで何を実装するのかは経験に依るところが大きい。

→**使いこなすには**オブジェクトを考える**訓練が必要**

今の段階で、どんなオブジェクトを作ればよいか思いつかなくても仕方がない。自主練習するしかない。

オブジェクト指向プログラミング

利点と欠点

利点

可読性が向上する : オブジェクト単位でコードが記述されるため
品質が保証しやすい : オブジェクト単位で動作検証ができるため
転用がしやすい : オブジェクトを流用できるため

欠点

人員確保 : オブジェクト指向を使いこなせる人員確保が難しい
再利用の手間 : 流用するためには部品化を見据えた設計開発が必要

オブジェクト指向プログラミング

まとめ

- オブジェクト指向プログラミングとは
現実世界のモノ等のオブジェクト単位でプログラムを開発していく考え方のことである.
- 修得には訓練が必要である.
- その前に, Javaの文法や使い方の理解が必要である.
→本講義ではまずJava文法が使いこなせるようになる
ところを目指す. その後オブジェクト指向ができる
ように訓練を行っていく (恐らくプロIVで).


本日の目次

- 復習
- クラスとインスタンス 1
- オブジェクト指向プログラミング
- **クラスとインスタンス 2**
- Javaのメモリ領域
- 命名規則
- 修飾子


クラスとインスタンス

this.とは

[インスタンス]. [フィールドorメソッド]
→ クラスの外からアクセスする書き方

	Monster m1 = new Monster(); m1.name = "スライミ"; // m1の名前を設定	Main.java
---	--	-----------

this.[フィールドorメソッド]
→ クラスの中からアクセスする書き方

	int hp = 100; public void damage(int attacked){ this.hp -= attacked; }	Monster.java
---	---	--------------

クラスとインスタンス

this.とは

[インスタンス].[フィールドorメソッド]
→ クラスの外からアクセスする書き方

this. [フィールドorメソッド]
→ クラスの中からアクセスする書き方

this.hpで自インスタンスのhpフィールドにアクセス

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    public void damage(int attacked){  
        this.hp -= attacked;  
    }  
}
```

ダメージを受けてHPが減少するという処理を実装

クラスとインスタンス

this.を付ける場合と付けない場合

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    public void damage(int attacked){  
        int hp = 50;  
        hp -= attacked;  
        this.hp -= attacked;  
        System.out.println("this.hp=" + this.hp);  
        System.out.println("hp=" + hp);  
    }  
}
```

thisを付けると必ず自インスタンスのフィールドにアクセスされる。

thisを付けないとローカル変数にアクセスされることがある。
(同名の変数が定義されている場合)

this.hp=80
hp=30

attacked=20の
場合の出力

クラスとインスタンス

this.を付ける場合と付けない場合

- ① 自インスタンスのフィールドにアクセスするとき
→ **必ずthis.**を付ける. (曖昧性の回避)
- ② メソッドのローカル変数にアクセスするとき
→ **this.**を付けない.

```
public class Monster{  
    int hp = 100;  
  
    public void damage(int attacked){  
        int hp = 50;  
    }  
}
```

クラスとインスタンス

練習問題 3 : this.とコンストラクタ

モンスターにエンカウント（遭遇）した際の定型文出力メソッドencount()を実装したい。空欄①②を埋めよ。

```
public class Monster{  
    int hp = 100;  
    String name = "スライミ";  
    public void encount(){  
        System.out.println( ① + "が現れた！");  
        System.out.println("HP:" + ②);  
    }  
}
```

スライミが現れた！
HP:100

クラスとインスタンス

コンストラクタ

下記のように、インスタンスを生成するたびに、フィールドを設定するのは面倒である。
そんな時は**コンストラクタ**を定義する。

```
public static void main(String[] args){  
    Monster m1 = new Monster();           // インスタンスの生成  
    m1.name = "スライミ";                 // m1 の名前を設定  
    m1.hp = 100;                           // m1 のHP を設定  
  
    Monster m2 = new Monster();           // インスタンスの生成  
    m2.name = "キメラン";                 // m2 の名前を設定  
    m2.hp = 1500;                         // m2 のHP を設定  
}
```

クラスとインスタンス

コンストラクタ

コンストラクタを定義するとスッキリ記述できる.

定義前

```
public static void main(String[] args){  
    Monster m1 = new Monster();           // インスタンスの生成  
    m1.name = "スライミ";                 // m1 の名前を設定  
    m1.hp = 100;                           // m1 のHPを設定  
    Monster m2 = new Monster();           // インスタンスの生成  
    m2.name = "キメラン";                 // m2 の名前を設定  
    m2.hp = 1500;                         // m2 のHPを設定  
}
```

定義後

```
public static void main(String[] args){  
    Monster m1 = new Monster(100, "スライミ");  
    Monster m2 = new Monster(1500, "キメラン");  
}
```


クラスとインスタンス

コンストラクタ

コンストラクタとは、インスタンスが生成された直後に実行される処理を記述する場所のことである。

定義方法は `public [クラス名]([引数], ...){}` である。

```
public class Monster{  
    int hp = 100;  
    String name = "";  
  
    public Monster(int hp, String name){  
        this.hp = hp;  
        this.name = name;  
    }  
}
```

コンストラクタの定義

初期化処理等を記述する

クラスとインスタンス

コンストラクタ

newの時に引数を渡すと、**コンストラクタ**に引数が送られるので、ここで初期値として設定する処理を書いておくことで、毎回の代入処理を記述しなくてよくなる。

```
public static void main(String[] args){  
    Monster m1 = new Monster(100, "スライミ");  
}
```

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    public Monster(int hp, String name){  
        this.hp = hp;  
        this.name = name;  
    }  
}
```

クラスとインスタンス

コンストラクタのオーバーロード（多重定義）

コンストラクタもオーバーロードできる。
コンストラクタの中で別コンストラクタを呼び出せる。

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    public Monster(int hp, String name){  
        this.hp = hp;  
        this.name = name;  
    }  
    public Monster(String name){  
        this(100, name);  
    }  
}
```

コンストラクタ 1 の定義

コンストラクタ 2 の定義

this(...)で別コンストラクタを呼び出せる
この場合、名前だけ設定するコンストラクタを呼ぶ
とHPは100で初期化される。

クラスとインスタンス

練習問題 4 : this.とコンストラクタ

そういえば練習問題 2 でapというフィールドを用意したことを忘れていた. apの初期値もコンストラクタで設定できるように, 3引数のコンストラクタを追加しよう.



クラスとインスタンス

練習問題 5 : 武器 (Weapon) クラスを作ってみよう

モンスターが装備する武器 (Weapon) を示すクラスを作成せよ. フィールドはString型のnameとint型のapとし, 両方を初期化するコンストラクタを明示すること.

クラスとインスタンス

クラス型のフィールド, クラス型の引数, 戻り値

クラス型のフィールドを定義することもできる.

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    int ap;  
    Weapon soubi;  
}
```

Weapon型のフィールド

クラスとインスタンス

クラス型のフィールド, クラス型の引数, 戻り値

soubiの初期値はnullなので, 装備を変更できるようにする.

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    int ap;  
    Weapon soubi;  
  
    public void setSoubi(Weapon soubi){  
        this.soubi = soubi;  
    }  
    public Weapon getSoubi(){  
        return this.soubi;  
    }  
}
```

Weapon型の引数

Weapon型の戻り値

クラスとインスタンス

クラス型のフィールド, クラス型の引数, 戻り値

呼び出し側の一例

```
public class Main{
    public static void main(String[] args){
        Monster m1 = new Monster(200, "スライミ", 10);
        Weapon weapon = new Weapon("ただの棒", 5);
        m1.setSoubi(weapon);
        Weapon m1Soubi = m1.getSoubi();
        System.out.println(m1.name + "の装備は" + m1Soubi.name);

        // こういう書き方もある
        m1.setSoubi(new Weapon("銅の剣", 15));
        System.out.println(m1.name + "の装備は" + m1.getSoubi().name);
    }
}
```


クラスとインスタンス

クラス型のフィールド, クラス型の引数, 戻り値

呼び出し側の一例

```
public class Main{  
    public static void main(String[] args){  
        Monster m1 = new Monster(200, "スライミ", 10);  
        Weapon weapon = new Weapon("ただの棒", 5); ①  
        m1.setSoubi(weapon);  
        Weapon m1Soubi = m1.getSoubi(); ②  
        System.out.println(m1.name + "の装備は" + m1Soubi.name); ③  
    }  
}
```

① : Weaponインスタンスの生成 (ただの棒) → m1の装備に設定

② : m1の装備を取得しm1Soubiインスタンスに代入
(①のweaponインスタンスと同じ)

③ : m1Soubi.nameで装備の名前フィールドを表示

クラスとインスタンス

クラス型のフィールド, クラス型の引数, 戻り値

呼び出し側の一例

```
public class Main{
```

④ : メソッドの引数で直接インスタンス生成を行うことも可能である.
(`new Weapon("銅の剣", 15)`が名もなきWeaponインスタンスという扱い)

⑤ : フィールド変数に直接アクセスすることも可能である.
(`m1.getSoubi()`が名もなきWeaponインスタンスという扱い)

// こういう書き方もある

```
m1.setSoubi(new Weapon("銅の剣", 15));
```

④

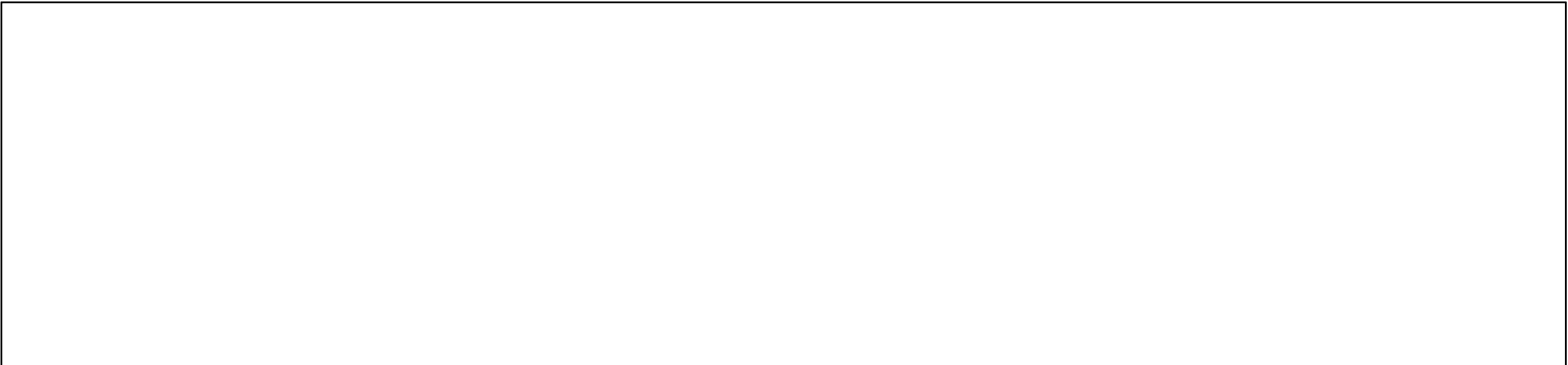
```
System.out.println(m1.name + "の装備は" + m1.getSoubi().name);
```

⑤

クラスとインスタンス

練習問題 6 : 武器 (Weapon) クラスをMonsterに装備させよう.

メインメソッドからボスモンスターインスタンス[boss]を
 > HP [15000], AP [300], NAME[リュウノウ]
で生成せよ. その後, bossに武器インスタンス[sword]を
 > NAME[魔王の剣], AP[150]
で生成し, 装備せよ.



本日の目次

- 復習
- クラスとインスタンス 1
- オブジェクト指向プログラミング
- クラスとインスタンス 2
- **Javaのメモリ領域**
- 命名規則
- 修飾子

Javaのメモリ領域

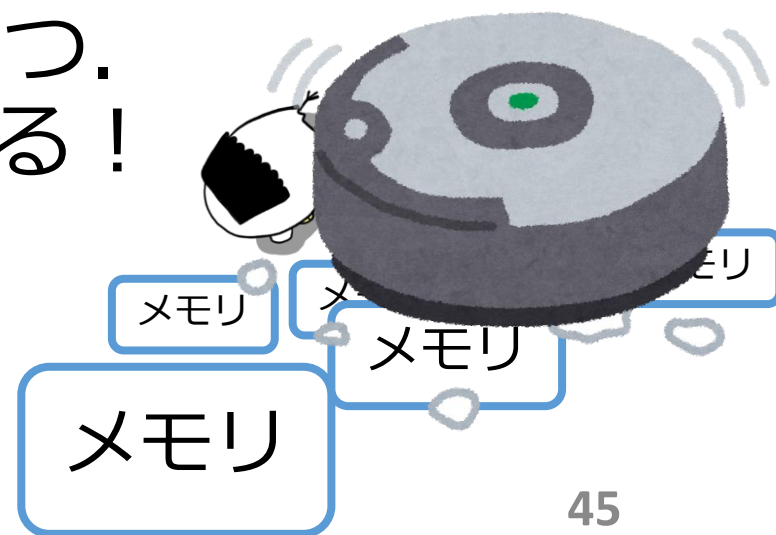
第一回の講義で、Javaにはガーベージコレクションという機能があり、人がメモリ管理を行わなくてもJVM（Java仮想マシン）が自動でメモリ管理を行ってくれると話した。

しかし、

メモリ管理の概要を知る。

→今後のJava構文理解に非常に役に立つ。

→メモリ管理の概要を簡単に説明する！



Javaのメモリ領域

Javaプログラムのデータは、メモリ内のスタック領域、ヒープ領域というところに格納される。

基本型（プリミティブ型）の場合

```
byte bVal = 1;  
int iVal = 10;  
double dVal = 100.0;
```

bValは190番地から1byte
iValは200番地から4byte
dValは210番地から8byte

スタック領域

01011100 bVal = 1
1100100101000110
iVal = 10
0101010000111011
dVal = 100.0

ヒープ領域

0101110010101000
1100100101000110
1111011111011010
1010101111110011
0101010000111011
1011010101010101
1101100011010101

Javaのメモリ領域

Javaプログラムのデータは、メモリ内のスタック領域、ヒープ領域というところに格納される。

インスタンスの場合

```
Monster m1 = new Monster();
```

- ① : Monsterが入る領域をヒープ領域に確保する.
- ② : 参照値を格納する領域をスタック領域に確保する.
- ③ : ②の領域に①の参照値を格納する.

スタック領域

```
0101110010101000
1100100101000110
1111011111011010
1010101111110011
m1 = 195 00111011
1011010101010101
1101100011010101
```

ヒープ領域

```
0101110010101000
195番地 0101000110
Monsterが入る領域
  hp
  name
  ap
1101100011010101
```

Javaのメモリ領域

Javaプログラムのデータは、メモリ内のスタック領域、ヒープ領域というところに格納される。

配列の場合

```
int[] arr = new int[5];
```

- ① : `int[5]`が入る領域をヒープ領域に確保する。
- ② : 参照値を格納する領域をスタック領域に確保する。
- ③ : ②の領域に①の参照値を格納する。

スタック領域

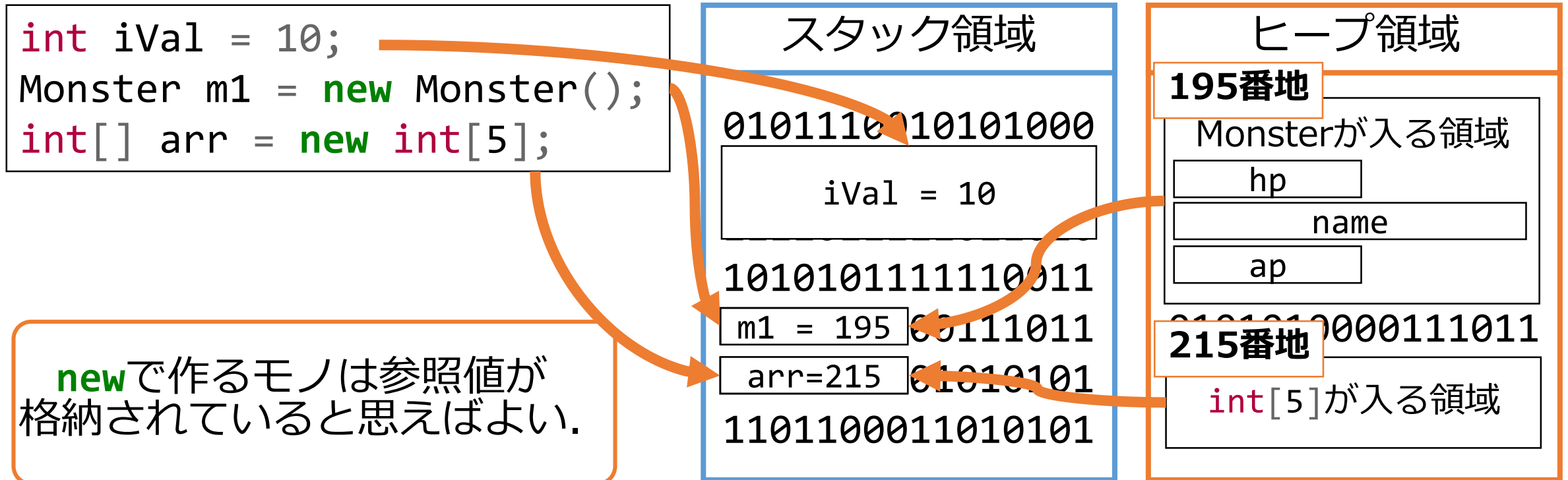
0101110010101000
1100100101000110
1111011111011010
1010101111110011
arr=215 00111011
1011010101010101
1101100011010101

ヒープ領域

0101110010101000
215番地 00101000110
int[5]が入る領域
arr[0]=0 arr[1]=0
arr[2]=0 arr[3]=0
arr[4]=0
1101100011010101

Javaのメモリ領域

基本型（プリミティブ型）は変数領域に値が直接入る。
参照型（インスタンスや配列）は変数領域に参照値が入る。

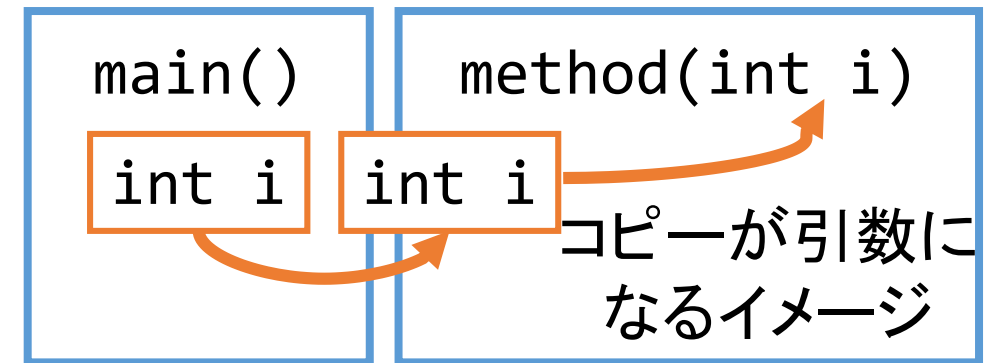


Javaのメモリ領域

int型等, 通常型の時

メソッドに引数で値を送るとき、
実物は送らず値だけを送る

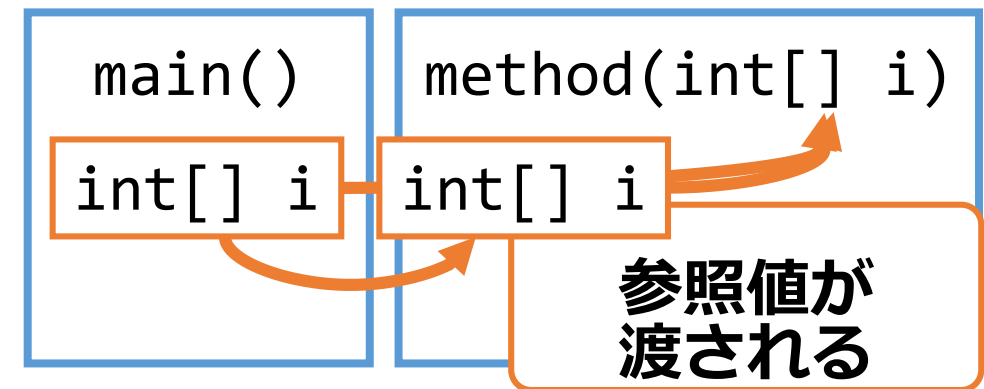
→メソッド内で値を編集しても、
呼び出し元変数は変わらない。



配列やインスタンス等, 参照型の時

メソッドに引数で値を送るとき、
実物をそのまま送るイメージ

→メソッド内で値を編集すると、
呼び出し元変数も変更される。



Javaのメモリ領域

従って、インスタンスを代入した場合も中身はコピーされない点には注意が必要である。

```
public class Main{  
    public static void main(String[] args){  
        Monster m1 = new Monster(200, "スライミ", 10);  
        Monster m2 = m1;  
  
        m2.name = "ドラゴヌ";  
        System.out.println("m1.name = " + m1.name);  
    }  
}
```

m2のnameに設定したのに、
m1.nameもドラゴヌになる。

m2 = m1;だと、m1の参照値が
m2に代入されるので、m2の参照
先はm1と同じとなる。

m1.name = ドラゴヌ

Javaのメモリ領域

練習問題 7 : 引数に送られる値

次のプログラムの出力①～③を予測せよ.

```
public static void main(String[] args){
    // Monster(int hp, String name, int ap){}
    Monster m1 = new Monster(200, "スライミ", 10);
    Monster m2 = m1;
    System.out.println(m1.name+", "+m1.hp+", "+m1.ap); //①
    shugyo(m1);
    System.out.println(m1.name+", "+m1.hp+", "+m1.ap); //②
    shugyo(m2);
    System.out.println(m1.name+", "+m1.hp+", "+m1.ap); //③
}

public static void shugyo(Monster m){
    m.hp += 10; //修行によりHP アップ
    m.ap += 5;  //修行によりAP アップ
}
```

本日の目次

- 復習
- クラスとインスタンス 1
- オブジェクト指向プログラミング
- クラスとインスタンス 2
- Javaのメモリ領域
- **命名規則**
- 修飾子

命名規則

変数, メソッド, クラスと一通りの説明を終えたので命名規則の一部を紹介する.

まず, Javaにおける命名の前提として以下がある.

- 先頭文字に数字は使えない.
- 文字数制限はない.
- 大文字と小文字は区別される.
- 予約語は使えない.

命名規則

クラス名

クラスの命名にはPascal記法を用いる。

Pascal記法は、以下3点の規則で成り立つ。

- ・ 先頭を大文字にする。
- ・ それ以外は小文字にする。
- ・ 単語の区切りは大文字にする。

例えば、

- ・ ArrayList
- ・ HashMap
- ・ SimpleDateFormat

命名規則

メソッド名, 変数名

メソッドと変数の命名にはcamelCase記法を用いる.
camelCase記法は, 以下3点の規則で成り立つ.

- 先頭を小文字にする.
- それ以外は小文字にする.
- 単語の区切りは大文字にする.

例えば,

- getTime()
- userName
- currentTimeMillis()

本日の目次

- 復習
- クラスとインスタンス 1
- オブジェクト指向プログラミング
- クラスとインスタンス 2
- Javaのメモリ領域
- 命名規則
- 修飾子

修飾子

これまでおまじないとしてきた**public**や**static**は、修飾子と呼ばれ、それぞれに意味がある。

publicと**private**は再来週以降のカプセル化で説明する。
本日は**static**と**final**に着目する。



修飾子

final

finalは、定数を宣言するための修飾子である。

```
final [変数型] [変数名] = [定数];  
> final int MAX_HP = 20000;
```

finalを付けると、変数のように読み出しはできるが、値の変更はできなくなる。

一般的に**final**定数の命名はアンダーバー区切りの全て大文字を使用する。

Cでいう#defineに近い。

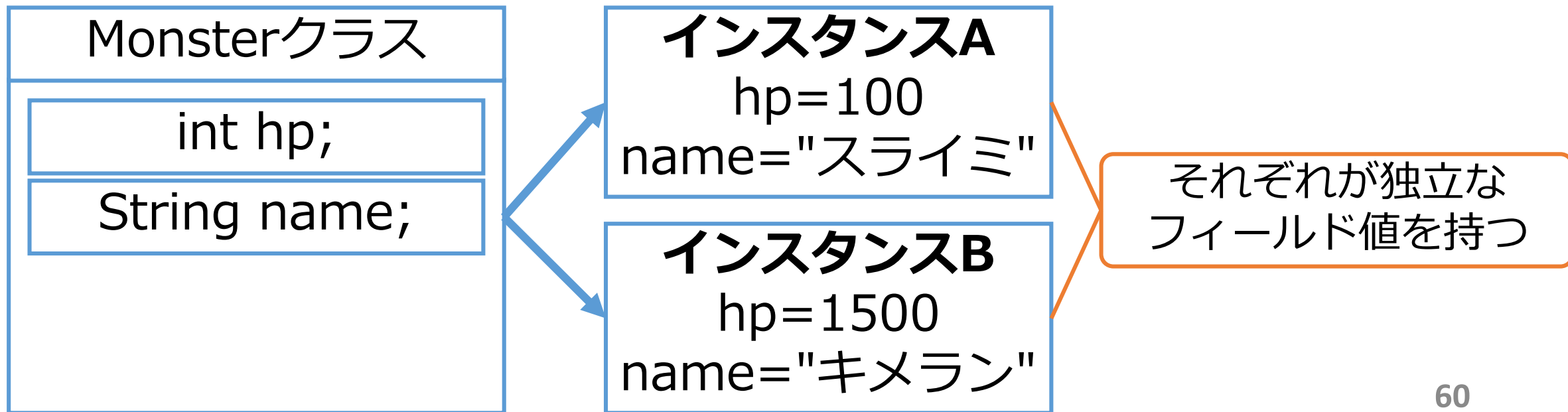


修飾子

static

staticは、静的メンバを宣言するための修飾子である。

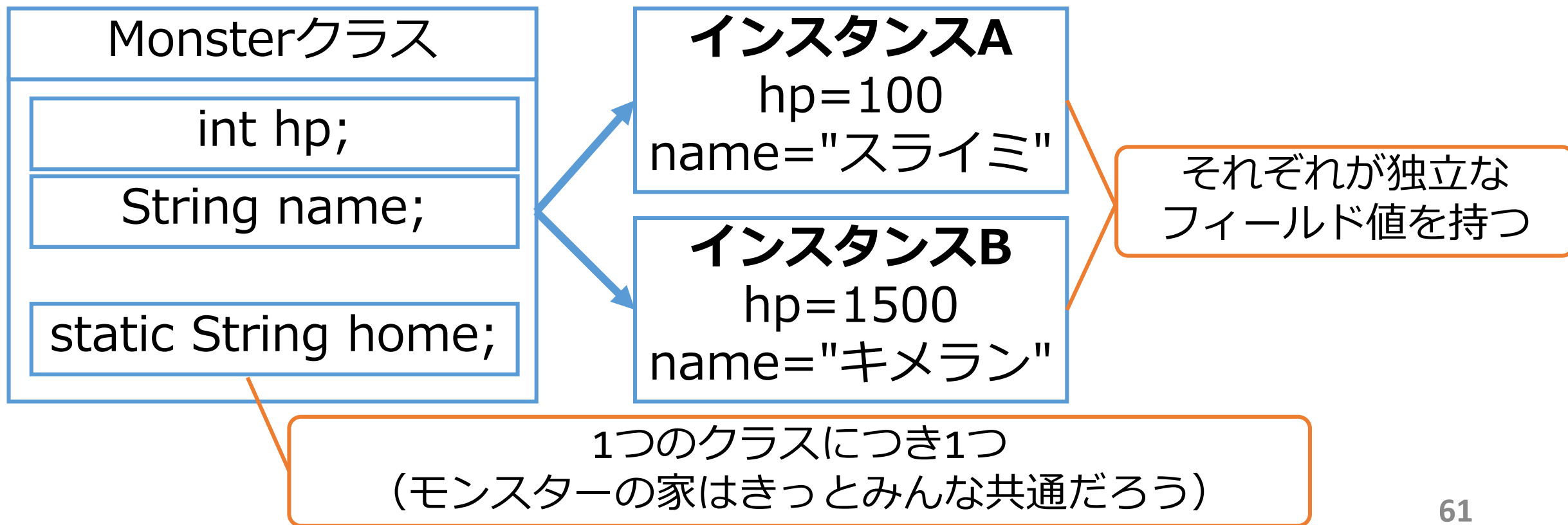
今回の講義で取り扱ってきたフィールドは全て**static**がついていない。**static**がついていない場合それぞれのインスタンスで独立なフィールド値を持つ。



修飾子

static

一方**static**がついているフィールドは静的メンバと呼ばれ、1つのクラスにつき1つ固有のフィールドしか持たない。



修飾子

static

アクセスするには、**[クラス名].[static変数名]**とするか、**[インスタンス].[static変数名]**とする。

```
public static void main(String[] args){  
    Monster m1 = new Monster(200, "スライミ", 10);  
    Monster m1 = new Monster(1500, "キメラン", 50);  
    Monster.home = "魔王の城";  
    System.out.println(Monster.home);  
    m1.home = "魔物の巣窟";  
    System.out.println(m2.home);  
}
```

[クラス名].[変数名]で
アクセスすることが**標準**

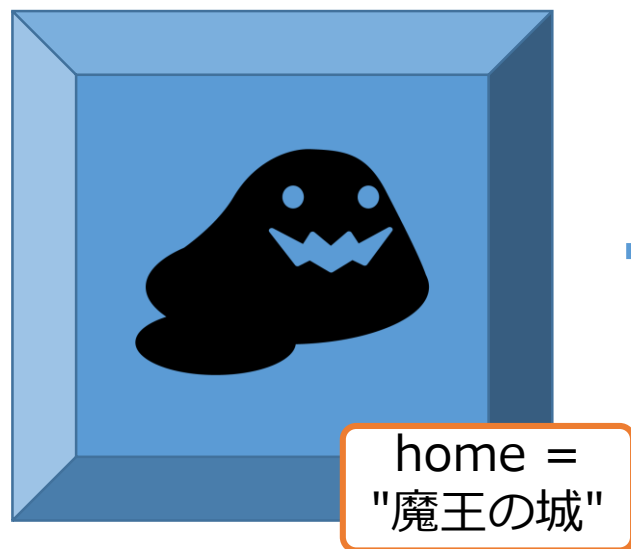
m1.homeを編集しても、homeは
唯一なのでm2.homeも変わる。
勿論、Monster.homeも変わる。

魔王の城
魔物の巣窟

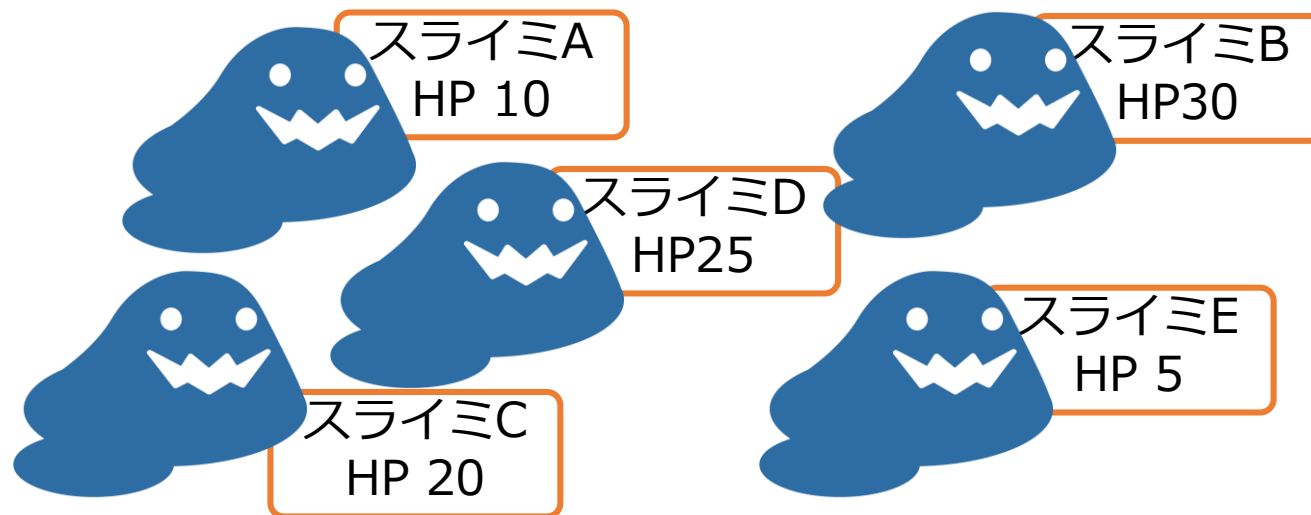
修飾子

static

クラス（金型）



インスタンス（量産されるイメージ）



staticフィールドは
クラスが唯一保持

```
public class Monster{  
    int hp = 100;  
    String name = "";  
    static String home = "";  
}
```

一般フィールドは個々
のインスタンスが保持

修飾子

static

staticがついている**メソッド**も定義することができる。

staticメソッドもクラスにつき唯一であるため、メソッド内から各インスタンスのフィールドにアクセスすることはできない。

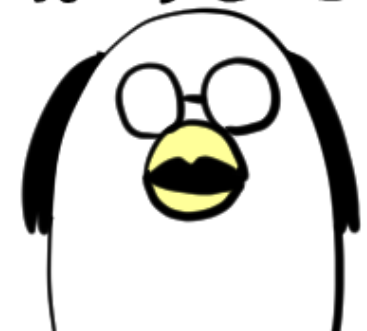
staticは、クラスにつき唯一のフィールドorメソッドとなるだけでなく、唯一なので、**わざわざnewしなくても呼び出せる**という特徴がある。以下の場合で使われたりする。

- 色々なクラスから呼び出される共通関数を定義する場合
(Mathクラスのメソッドsqrt()やmax()とか)
- 共通定数を定義する場合
- 複数インスタンス間で値を共有するフィールドを作る場合

本日のまとめ

- オブジェクト指向の基本であるクラスとインスタンスについて学習した
- オブジェクト指向の考え方についても少し触れた.
- Javaのメモリ領域の概要を理解し, プリミティブ型とインスタンスの引数受け渡され方の違いを学習した.
- Javaの命名規則を学習した.
- finalとstaticについて学習した.

わかりました



次週予告

※次週以降も計算機室

本日の復習と，本日の内容に関する演習問題を実施する．



本日の提出課題

提出はWebClassで

課題 1

本日の授業を聞いて、
初めて知ったと思う内容を2点簡潔に述べよ。
「簡潔に述べよ」⇨「1, 2文程度で述べよ」と考えるとよい

課題 2

本日の授業を聞いて、
質問事項または**気になった点**を2点簡潔に述べよ。

課題 3

自分なりに、クラスとインスタンスの違いを説明せよ。