復習 2

クラス，インスタンス，継承，カプセル化

2018/4/17(火)　プログラミングIV 第二回

福井大学　工学研究科　情報・メディア工学専攻

長谷川達人

量が多いので，復習用に資料はあるが説明を省略することがある.

演習問題の解説

- HP上にアップロードした.
- 難所？はコメントしてあるので各自確認すること.
  - 今回も内容豊富なため，授業時間に余裕がないので.

- 学籍番号を入力すると，成績を表示する仕組みの導入を検討しているが，自分の成績がどうしても見られたくない人は，本日の感想に書いて下さい.

質問に対する回答
メモリについて

- メモリ領域の話はどこまでわかっていないと困るのか.
  - 具体的には，｛基本型，配列，インスタンス｝を｛メソッドに引数で送った時，代入でコピーした時｝に影響する.
  - 基本型はコピー先で変更しても，もとの変数は変わらない.
  - 配列とインスタンスはコピー先で変更すると，もとの配列とインスタンスの中身が変更される.

  - その理由が，先週のメモリの話である.

質問に対する回答

色々

- 変数の初期値は？
  - 数値型→0　boolean型→false　他→null
- 大きい型→小さい型キャストはいつ使う？
  - よくあるのは文字列→数値型
  - 安定動作を図りづらいので使わない方が良い.
- xorの実装にA^Bのようなビット演算子を．．．
  - バレたか．．それでも実装可能である.
- 編入生が授業についていくためは？
  - 本日までの部分をしっかり復習しておくこと.
  - 次週以降は完全に新しいお話なので.

質問に対する回答

- メソッド間で同じ変数を使うにはどうすればよいか
  - グローバル変数的なものがあるが非推奨である.
  - そもそもオブジェクト指向は，クラス間やメソッド間をなるべく独立できるように設計すべきである.
- メソッドが増えたときクラスを分けるべきか
  - クラスが保有すべきメソッドは，なるべくフィールドを操作するもの，そのオブジェクト固有のものであるべきである.
- methodA(int)とmethodA(int...)が同時に存在できるのか
  - できる．可変長でない方が優先される.

質問に対する回答

評価について

- たまたま同じコードになった場合コピペなの？
  - もちろんたまたま同じ場合は不正ではないので気にしなくて良い.
- 評価の配分はどのくらいか
  - まだ確定はしていないが下記の予定である.
    - 各回の課題：4割
    - 自由開発演習の評価：2割
    - 期末試験の成績：4割
- 自由開発ではどんなものを作ればよいのか
  - 何でも良いが適当すぎるものは評価が下がる.
  - 面白さ，開発難易度，アピール力　で総合評価予定

本講義の概要

前半		後半	
第1回	基本文法の復習	第9回	標準ライブラリ
第2回	**クラス～カプセル化の復習**	第10回	ファイル入出力
第3回	抽象クラス，インタフェース	第11回	デバッグ，インポート，高速化
第4回	ポリモーフィズム	第12回	オブジェクト指向
第5回	GUI：Canvas	第13回	自由開発演習 1
第6回	GUI：Layout，イベントリスナ	第14回	自由開発演習 2
第7回	スレッド，例外処理	第15回	自由開発演習発表会
第8回	ジェネリクス，コレクション	第16回	期末試験

何を開発するか，少しずつ考えておくこと

本日の目標

概要

Javaで最も重要な考え方の**オブジェクト指向**について，
クラス，インスタンス，継承，カプセル化に関する復習を行う．

目標

カプセル化したクラスを継承したオリジナルクラス
を実装でき，インスタンス化して利用までできる．

本日の提出課題

講義パート

課題を意識しながら
講義を聞くと良い.

課題 1

本日の授業を聞いて,
よくわかったと思う内容を2点簡潔に述べよ.

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1点以上簡潔に述べよ.

課題 3

感想（あれば）

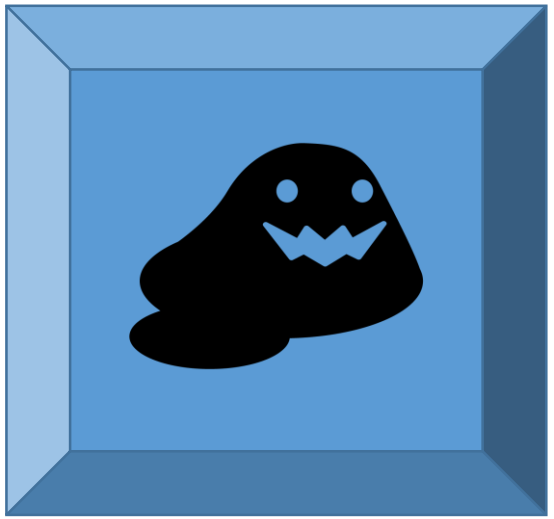
クラスとインスタンス

イメージとしては,

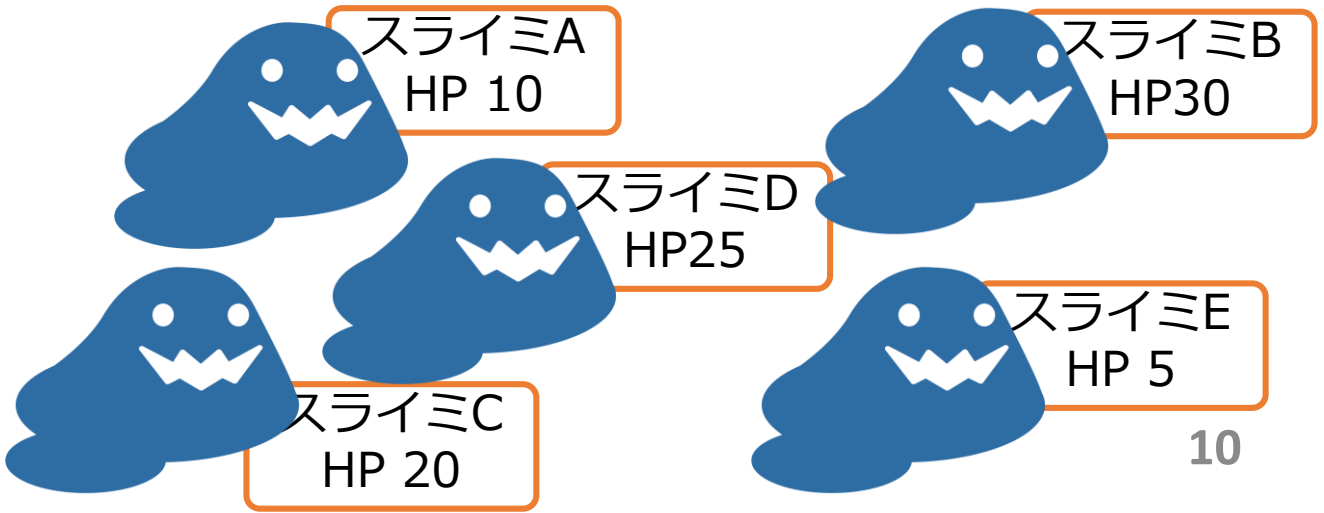
クラス　＝設計図や金型

インスタンス　＝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

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．

```
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);
        }
}
```

フィールド

メソッド

staticを付けない

左のような金型を事前に定義したもの＝クラス

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）

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

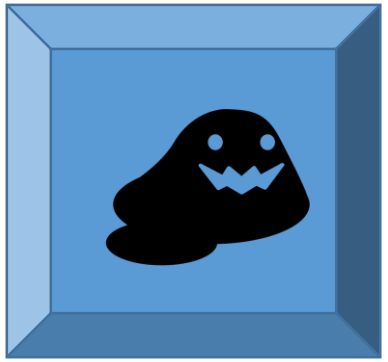
インスタンスの作成

[クラス名] [変数名] = new [クラス名]();

クラスとインスタンス

各フィールドはインスタンスごとに保持される

金型



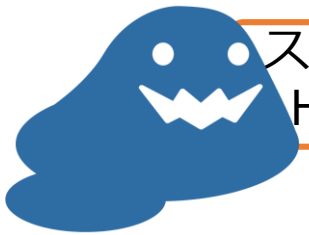
Monster
クラスの定義

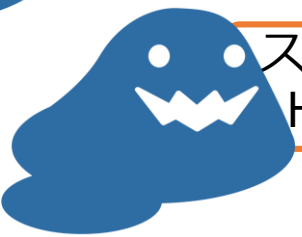
```
public class Monster{
	int hp = 100;
	String name = "";
	public int attack(){
		// 0～9までの乱数のダメージを返す
		return (int)(Math.random()*10);
	}
}
```

```
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
}
```

インスタンス内部には「.」でアクセスする

インスタンスの生成

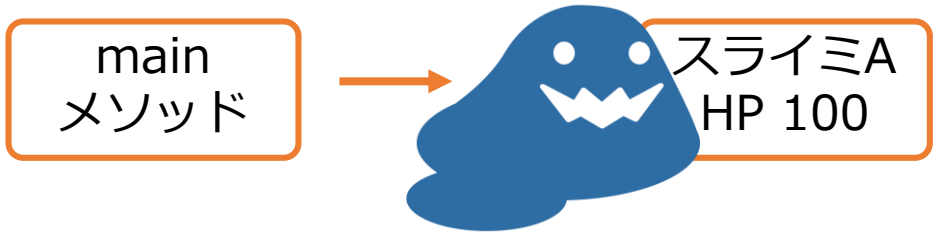




クラスとインスタンス

this.とは

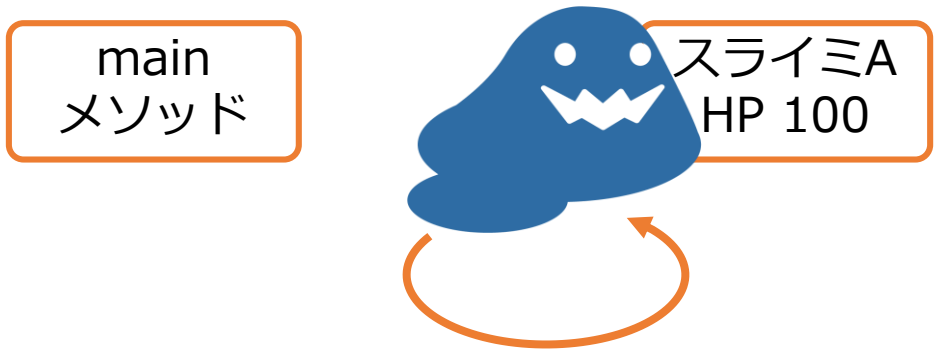
mainメソッドからインスタンス内のフィールドを操作する場合



Main.java

```
Monster m1 = new Monster();
// [インスタンス名].[フィールド名]でアクセス
m1.name = "スライミ"; // m1の名前を設定
```

自身のインスタンス内のフィールドを操作する場合



Monster.java

```
int hp = 100;
public void damage(int attacked){
    hp -= attacked;
}
```

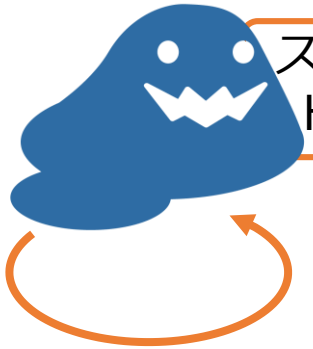
直接フィールド名を指定することもできるが…

クラスとインスタンス

this.とは

もし，同メソッド内でhpという変数名が使用されていた場合…

main
メソッド



スライミA
HP 100

Monster.java

```
int hp = 100;
public void damage(int attacked){
        int hp = 50;
        hp -= attacked;
        this.hp -= attacked;
}
```

そのままhpと書くとメソッド内の変数hpを意味する.

this.[フィールドorメソッド]とすることで確実に自身のフィールドを意味する.

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;
	}
}
```

①クラスのフィールド

②メソッドのローカル変数

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;
	}
}
```

戻り値がないメソッドみたい
＋メソッド名がクラス名と一致する

コンストラクタの定義

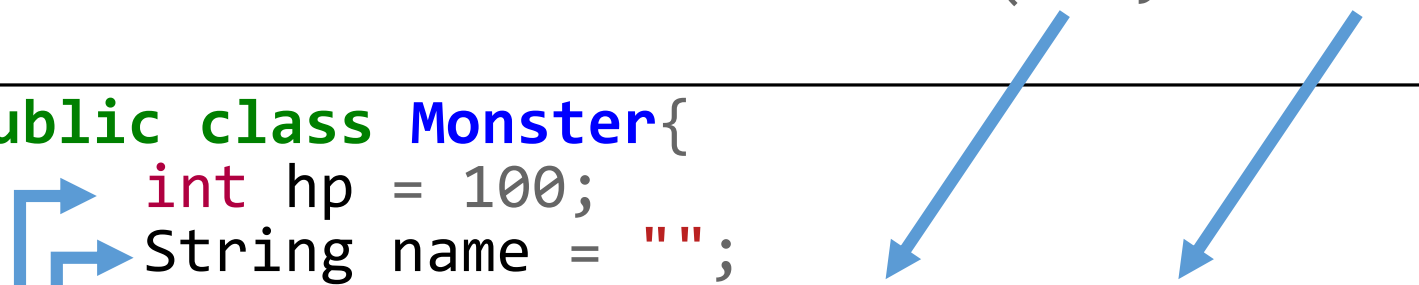
初期化処理等を記述する

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

クラスとインスタンス

練習問題１：クラスの定義

次の手順に沿って，Studentクラスを作成せよ.

1. Studentクラスを作成し，文字列型のname，数値型のscoreJp, scoreEnのフィールドを定義せよ.
2. nameを初期化するコンストラクタを定義せよ.
3. nameとscoreJp，scoreEnを初期化するコンストラクタをオーバーロードで定義せよ．ただしnameの初期化は２で定義したコンストラクタを使用すること.
4. scoreJpとscoreEnの平均点を返すメソッド double avg([引数なし])を定義せよ．staticは不要.

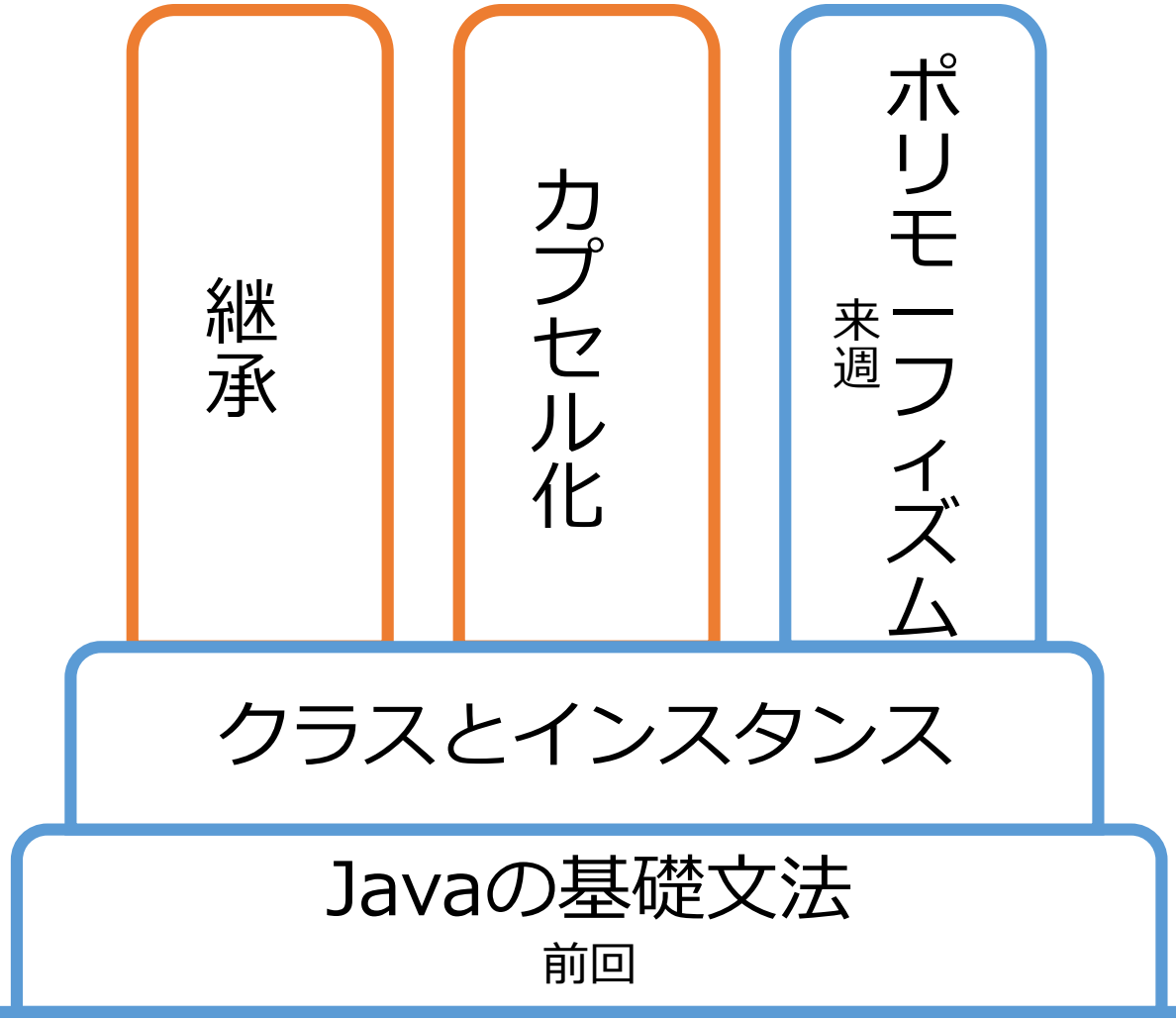
クラスとインスタンス

練習問題２：クラスの操作

Studentクラスを以下の手順で操作せよ.

1. Main.javaにmainメソッドを作成し，Studentインスタンスhaseを，コンストラクタを用いてnameを“はせがわ”で初期化して作成せよ.
2. コンストラクタを使わずに，haseのscoreJpを50に，scoreEnを70に変更せよ.
3. Studentインスタンスfukuをコンストラクタを用いてname=“ふくま”，scoreJp=80，scoreEn=90で初期化して作成せよ.
4. はせがわの平均点=60.0
   ふくまの平均点＝85.0　のようにコンソール出力せよ.

オブジェクト指向の3大要素



これらは，オブジェクト指向を行う際に役に立つ3つの考え方みたいなものである．

継承

Monsterクラスのインスタンスとしてボスモンスターを作ろうと思ったが，ボスはすごい魔法を使ってくるよう**少し改造したい**.

どうすればよいだろうか？
（Monsterインスタンスだとattack()メソッドしかない）

```
public class Monster{
	int hp = 0;
	String name = "";
	int ap = 0;
	public int attack(){
		return this.ap;
	}
}
```

継承

案１．Monsterクラスにmagicメソッドを実装する.

＞ボス以外のMonsterインスタンスもmagic()が使えてしまう.

```
public class Monster{
	int hp = 0;
	String name = "";
	int ap = 0;
	public int attack(){
		return this.ap;
	}
	public int magic(){
		System.out.println(this.name+"は魔法を詠唱した. ");
		return this.ap + (int)(Math.random()*10);
	}
}
```

継承

案２．Monsterクラスをコピペしてbossmonsterクラスを作る.

＞attack()や共通のフィールドに変更があったとき，両方直さなければならない.

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

```
public class BossMonster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
  public int magic(){
    System.out.println(this.name+"は魔法を詠唱した. ");
    return this.ap + (int)(Math.random()*10);
  }
}
```

継承

案３．継承（extends）を使ってBossMonsterクラスを作る.

Monsterクラスがコピペされている
前提と考えることができる.

extends 継承したいクラス名
をクラス定義時に指定する.

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

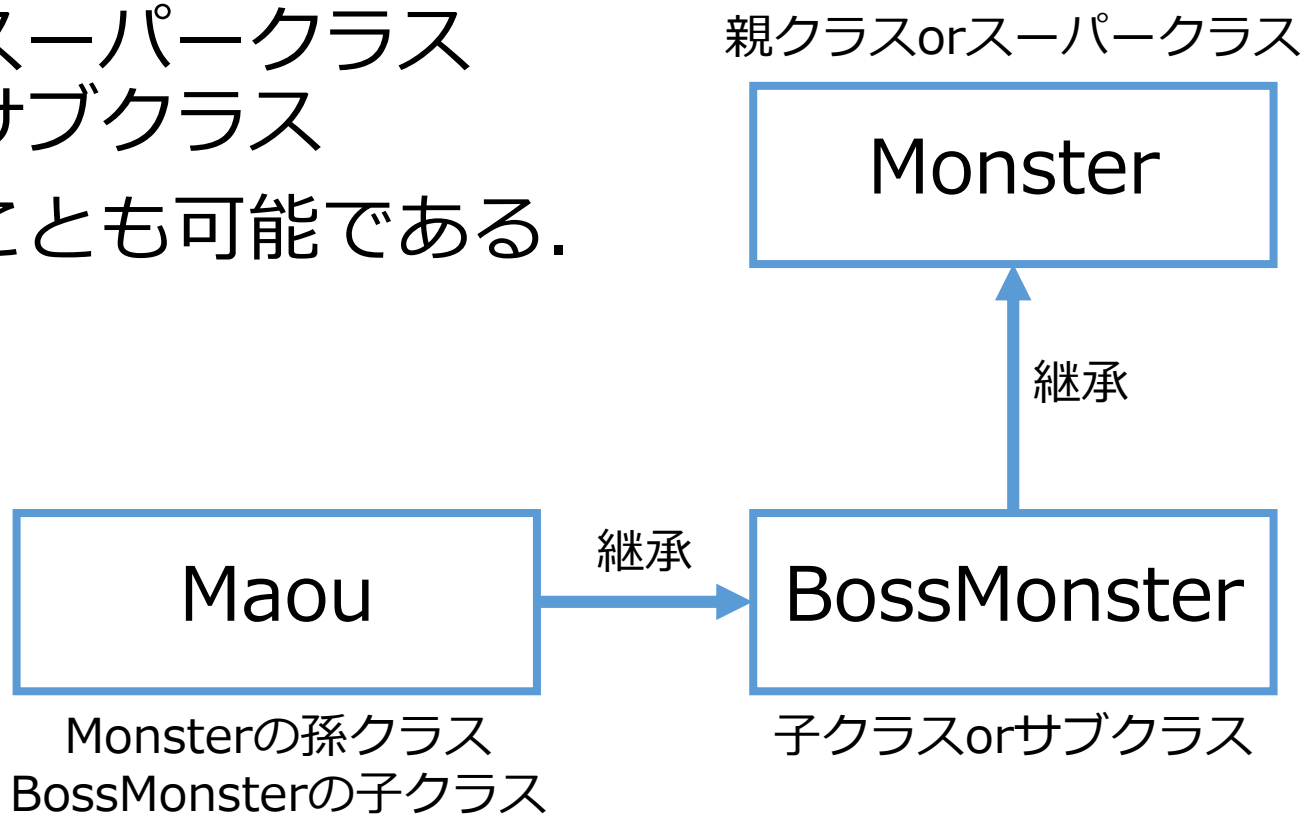
```
public class BossMonster extends Monster{

  public int magic(){
    System.out.println(this.name+"は魔法を詠唱した. ");
    return this.ap + (int)(Math.random()*10);
  }
}
```

継承

継承関係は右図のように図示する.

継承される側：**親クラス**orスーパークラス
継承する側　：**子クラス**orサブクラス

子クラスをさらに継承することも可能である.



継承

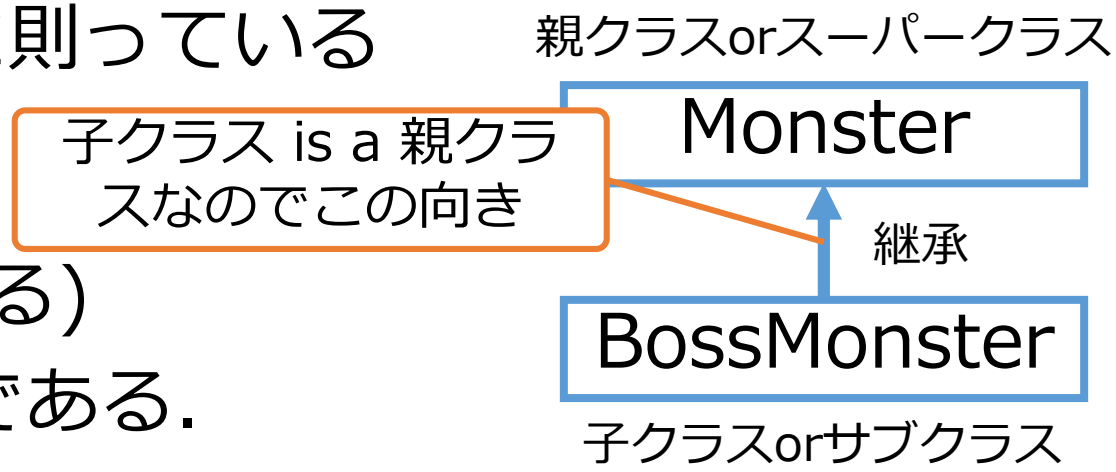
正しい継承とは，「is-aの原則」に則っている継承関係のことである．即ち，

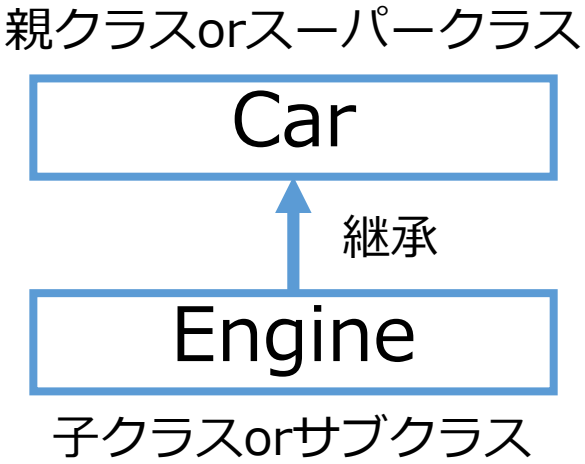
子クラス **is a** 親クラス.
(子クラスは親クラスの一種である)

が成立する継承関係が正しい継承である.

これが不成立となる場合，継承関係にしてはならない（三大要素の一つポリモーフィズムで不具合が起こってくるため）.

上　BossMonster is a Monster.　-> 成立
下　Engine is a Car.　-> 不成立
※エンジンは車の一種ではない.



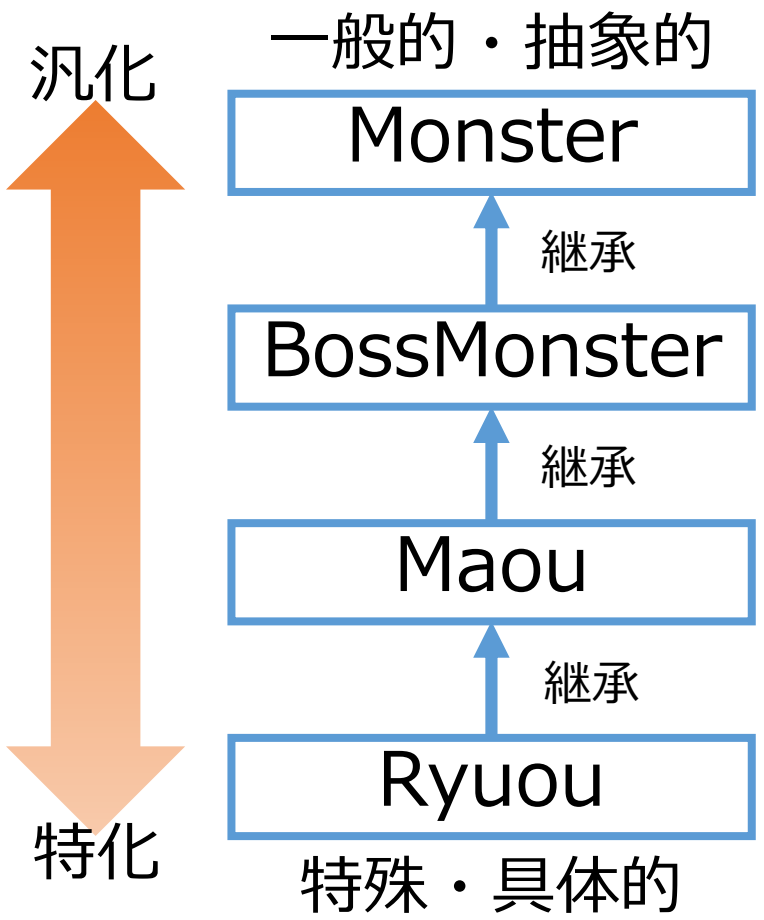


継承

継承関係は，汎化・特化の関係ともいえる．

右の例では，

- Monsterは全ての敵キャラ共通の項目（HPや名前）を管理する．
- BossMonsterは全てのボスキャラ共通の項目（攻撃時にAPが上昇等）を管理する．
- Maouは全ての魔王キャラ共通の項目（遭遇時のセリフ等）を管理する．
- Ryuouは魔王の中でもリュウオーに特化した処理や項目を管理する．



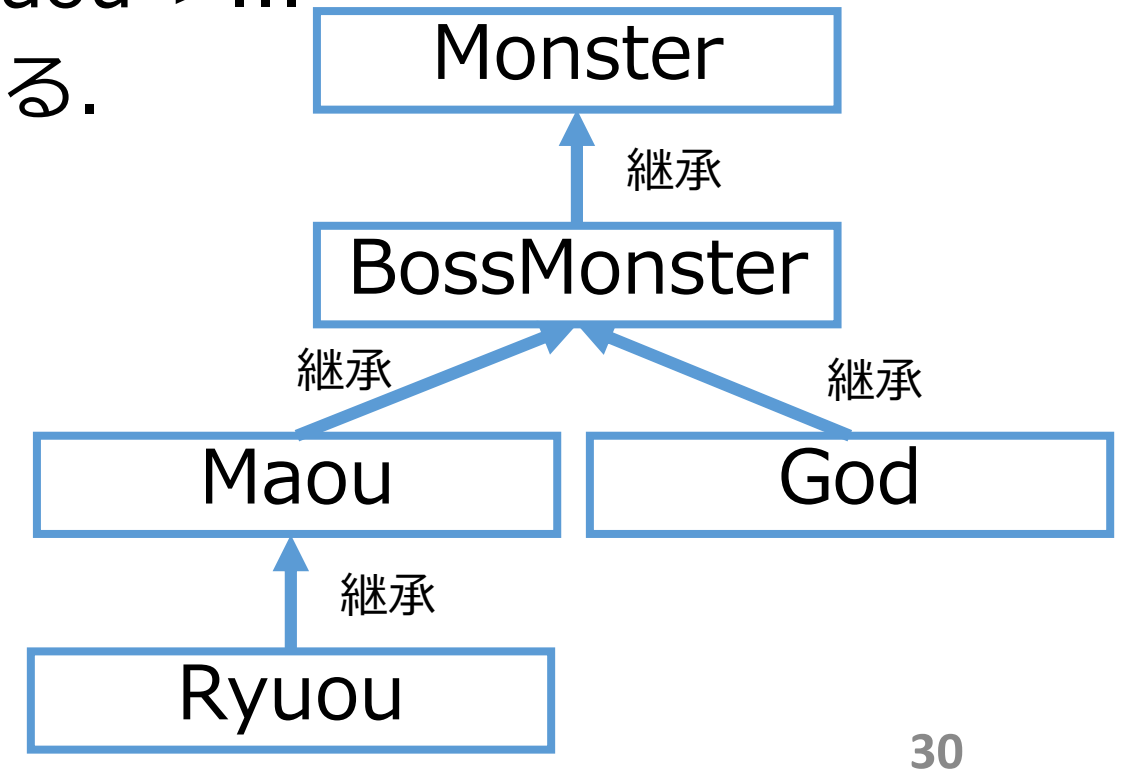
継承

・継承に限界はないのか？親クラスを複数持てるのか？

1．子クラスをどんどん継承していくことができる.
　＞Monster->BossMonster->Maou->...

2．複数の子クラスを持つことができる.
　＞BossMonster->Maou & God

3．複数の親クラスを持つことは（Javaでは）できない.
　＞Ryuou extends Maou, God はできない.



継承
オーバーライド

BossMonsterクラスを継承により実装したが，ボスは通常攻撃を行うごとに攻撃力（AP）が上昇するよう**少し改造したい**.

どうすればよいだろうか？
（attack()はMonsterクラスにあるが，BossMonsterのみattack()を改造したい．）

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

```
public class BossMonster extends Monster{
  public int magic(){
    System.out.println(this.name+"は魔法を詠唱した． ");
    return this.ap + (int)(Math.random()*10);
  }
}
```

継承
オーバーライド

案１．BossMonsterクラスにattack2()を実装する.

＞通常Monsterはattack()を呼び出し，BossMonsterは
attack2()を呼び出さなければならない.

＞＞ヒューマンエラーの原因になるのでは？

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

```
public class BossMonster extends Monster{
  public int magic(){
    System.out.println(this.name+"は魔法を詠唱した. ");
    return this.ap + (int)(Math.random()*10);
  }
  public int attack2(){
    this.ap += 5;
    return this.ap;
  }
}
```

こんな感じ

継承
オーバーライド

案２．オーバーライドを使ってattack()メソッドを上書きする.

＞通常MonsterもBossMonsterもattack()で統一できる.

＞＞ヒューマンエラーのリスクを減らせる.

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

```
public class BossMonster extends Monster{
  public int magic(){
    System.out.println(this.name+"は魔法を詠唱した. ");
    return this.ap + (int)(Math.random()*10);
  }
  public int attack(){
    this.ap += 5;
    return this.ap;
  }
}
```

親クラスと全く同じ
メソッドを再定義すること
＝オーバーライド

継承
オーバーライド

オーバーライドを使う時には@Override注釈をつける.

これは今からオーバーライドを書きますよという宣言である.

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

```
public class BossMonster extends Monster{
  @Override
  public int attack(){
    this.ap += 5;
    return this.ap;
  }
}
```

ここに書く

```
public class BossMonster extends Monster{

  public int attack(){
    this.ap += 5;
    return this.ap;
  }
}
```

どちらも正常に動く

継承
オーバーライド

オーバーライドを使う時には@Override注釈をつける.

注釈をつけることでヒューマンエラーのリスクを減らせる.

```java
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

```java
public class BossMonster extends Monster{
  @Override
  public int attackk(){
    this.ap += 5;
    return this.ap;
  }
}
```

たまたま書き間違えたとき

親クラスにattakk()がないとコンパイルエラー

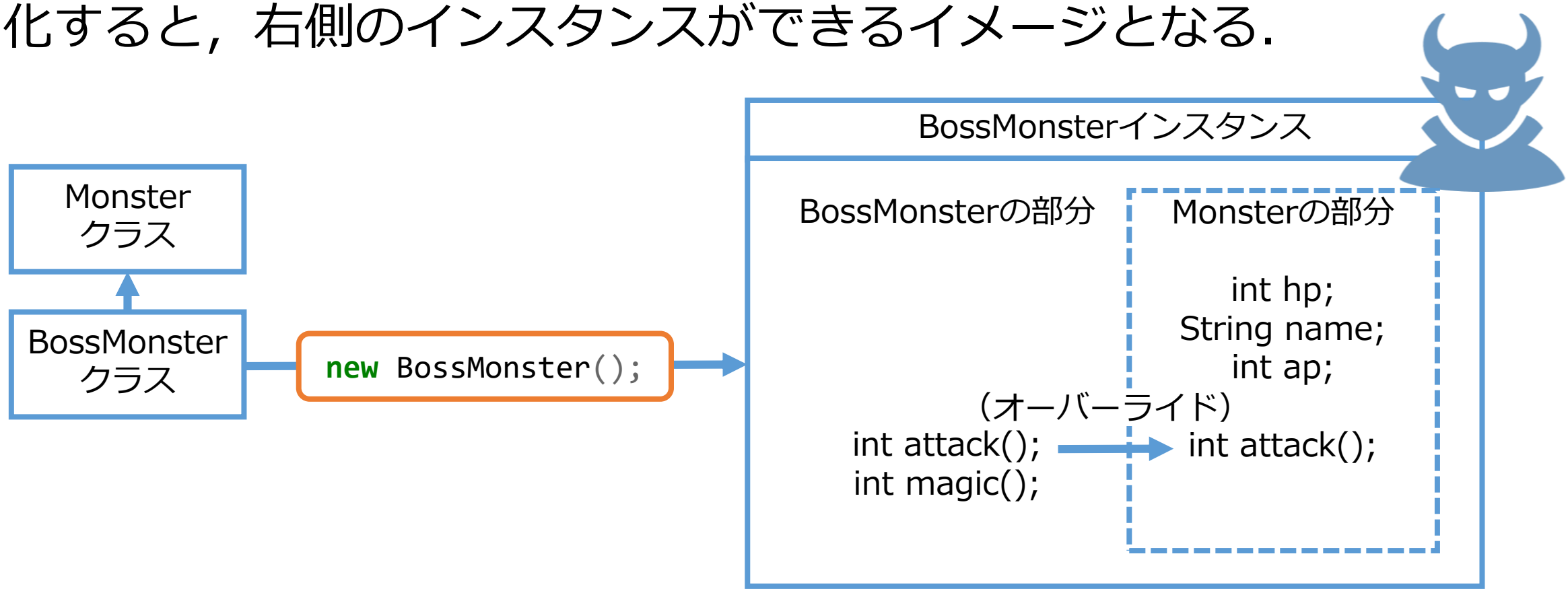
```java
public class BossMonster extends Monster{

  public int attackk(){
    this.ap += 5;
    return this.ap;
  }
}
```

attakk()という新メソッドと認識される

継承

継承とオーバーライドのイメージ

Monsterクラスを継承したBossMonsterクラスをインスタンス化すると，右側のインスタンスができるイメージとなる.

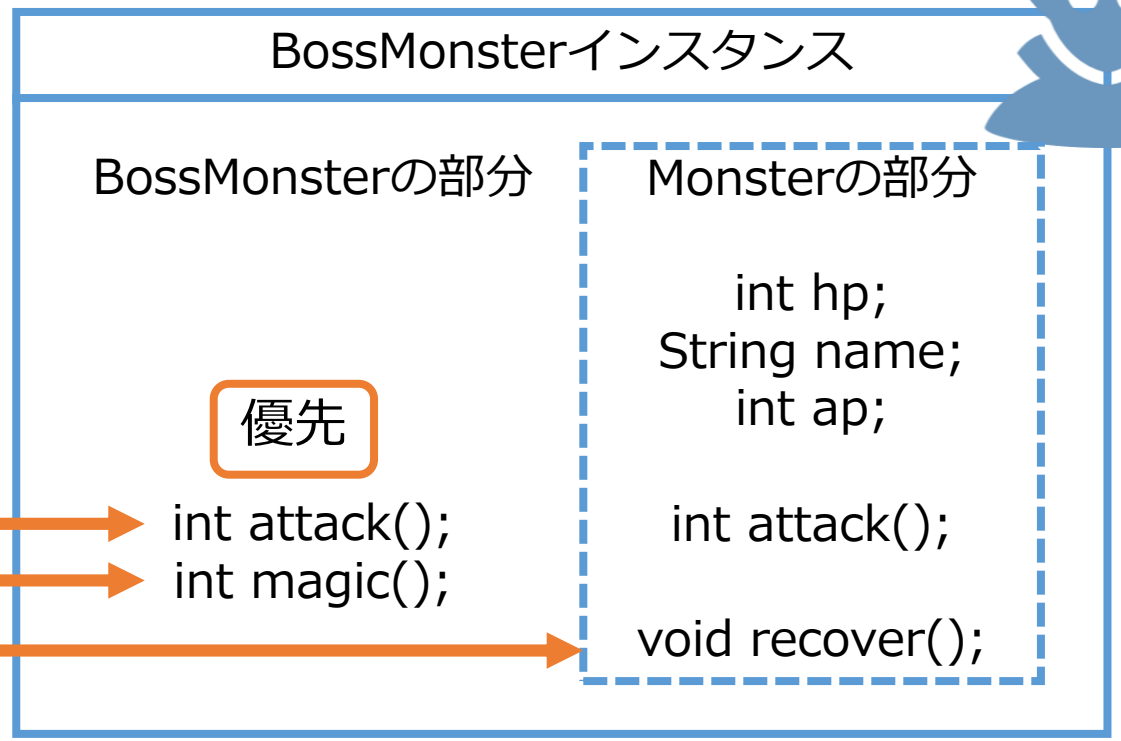


継承

継承とオーバーライドのイメージ

BossMonsterに実装されているメソッドを優先して使用する.
実装されていない場合はMonsterに実装されているメソッドを使用する.

```
public static void main(String[] args){
  BossMonster boss = new BossMonster();
  boss.attack();
  boss.magic();
  boss.recover();
}
```

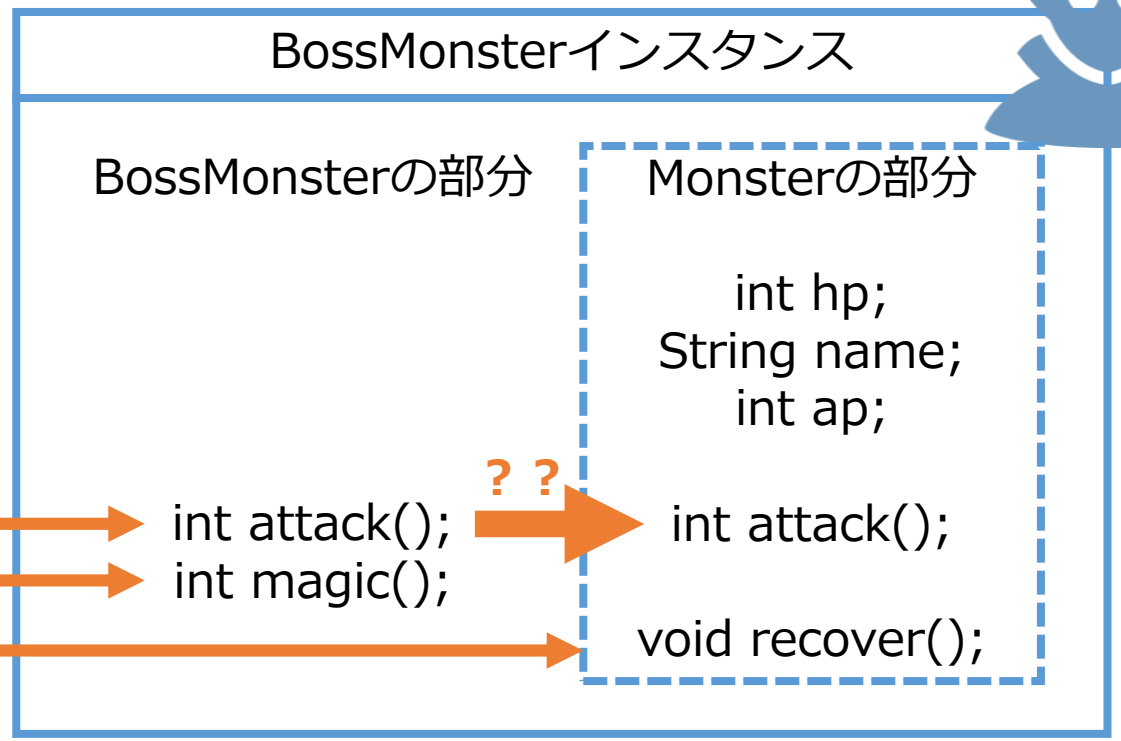


継承
親クラスのメソッドへのアクセス

下記の例の場合，Monster側のattack()にアクセスできないのか？

＞BossMonster内のメソッドからならアクセスできる.

```
public static void main(String[] args){
  BossMonster boss = new BossMonster();
  boss.attack();
  boss.magic();
  boss.recover();
}
```



継承
親クラスのメソッドへのアクセス

例えば，先ほどの例を考える．

もし，Monsterのattack()の仕様が変わった場合どうなる？

＞BossMonsterのattack()も修正が必要でバグの要因になり得る．

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;

  public int attack(){
    return this.ap+5;
  }
}
```

```
public class BossMonster extends Monster{



 @Override
  public int attack(){
    this.ap += 5;
    return this.ap+5;
  }
}
```

両方修正しなければならない
（将来的に忘れられるかも？）

継承
親クラスのメソッドへのアクセス

super.method()を使うことで，この問題を回避できる.

＞Monsterのattack()を修正すれば，BossMonsterから呼び出す
super.method()も修正される.

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;

  public int attack(){
    return this.ap+5;
  }
}
```

```
public class BossMonster extends Monster{


 @Override
  public int attack(){
    this.ap += 5;
    return super.attack();
  }
}
```

super.method()で
親クラスのメソッドを
呼び出せる

継承
オーバーライドのまとめ

オーバーライド：親クラスの全く同じメソッドを再定義すること

オーバーロード：同じメソッド名を多重定義すること（復習）

メソッドの直前には「@Override」注釈を記述する.

親クラスのメソッドやフィールドにアクセスする際には**super.**を付ける.

```
public class Monster{
  int hp = 0;
  String name = "";
  int ap = 0;
  public int attack(){
    return this.ap;
  }
}
```

オーバーライド

```
public class BossMonster extends Monster{
  @Override
  public int attack(){
    super.ap += 5;
    return super.attack();
  }
}
```

継承
コンストラクタ

次のプログラムの実行結果を予測してみよう.

```
public class Main {
	public static void main(String args[]){
		ClassB b = new ClassB();
	}
}
```

```
public class ClassB extends ClassA{
	public ClassB(){
		System.out.println("ClassB");
	}
}
```

```
public class ClassA {
	public ClassA(){
		System.out.println("ClassA");
	}
}
```

【出力】
ClassA
ClassB

ClassAコンストラクタも
実行されるのはなぜ？

継承
コンストラクタ

Javaでは全てのコンストラクタは，その先頭で必ず親クラスのコンストラクタを呼び出さなければならない.

```
public class Main {
	public static void main(String args[]){
		ClassB b = new ClassB();
	}
}
```

```
public class ClassB extends ClassA{
	public ClassB(){
		System.out.println("ClassB");
	}
}
```

```
public class ClassA {
	public ClassA(){
		System.out.println("ClassA");
	}
}
```

ここで暗黙的に親クラスの引数無しコンストラクタが呼び出されている

継承
コンストラクタ

親クラスのコンストラクタを明示的に呼び出すには，
super([引数]);と記述する．

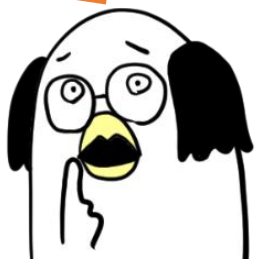
```
public class Main {
	public static void main(String args[]){
		ClassB b = new ClassB();
	}
}
```

```
public class ClassB extends ClassA{
	public ClassB(){
		super();
		System.out.println("ClassB");
	}
```

```
public class ClassA {
	public ClassA(){
		System.out.println("ClassA");
	}
}
```

何も書かなかった場合，
ここに暗黙の**super();**を
自動で挿入していた．

自クラスのコンストラクタ
を呼び出すには**this();**を
使っていましたね（復習）．



継承
コンストラクタ

ただし，暗黙の**super()**は**引数無しコンストラクタしか自動挿入されない**．したがって，以下のケースでは明示する必要がある．

```
public class Main {
	public static void main(String args[]){
		ClassB b = new ClassB();
	}
}
```

```
public class ClassB extends ClassA{
	public ClassB(){
		System.out.println("ClassB");
	}
}
```

```
public class ClassA {
	public ClassA(int a){
		System.out.println("ClassA");
	}
}
```

暗黙の**super();**を自動挿入しようとしても，親クラスには引数無しコンストラクタがなかった！！

今回の場合，**super(100);**等を手動で明記する必要がある．

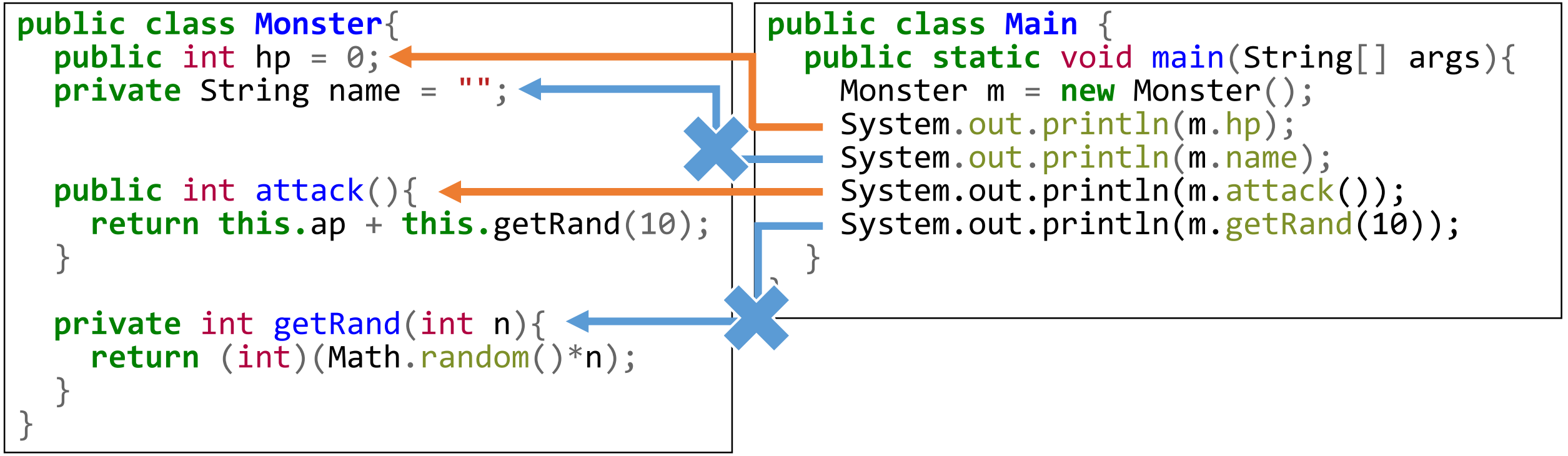
継承

練習問題３：特進生徒

Studentクラスを継承し，特進生徒であるSpecialStudentクラスを，次の手順で作成せよ.

1. SpecialStudentクラスを作成しStudentを継承せよ．また，独自のフィールドscoreMathを数値型で定義せよ.
2. Student同様に1つ目のコンストラクタはnameを初期化するものを定義せよ．2つ目のコンストラクタはscoreMathも初期化するものとせよ．なお，scoreMath以外の初期化は**Studentクラスのコンストラクタ**を利用すること.
3. avg()をオーバーライドしscoreMathも含め平均値を算出するようにせよ.
4. Main.javaのmainメソッドでSpecialStudentインスタンスを作成し，avg()の動作確認を実施せよ.

カプセル化



```
public class Monster{
  public int hp = 0;
  private String name = "";


  public int attack(){
    return this.ap + this.getRand(10);
  }

  private int getRand(int n){
    return (int)(Math.random()*n);
  }
}
```

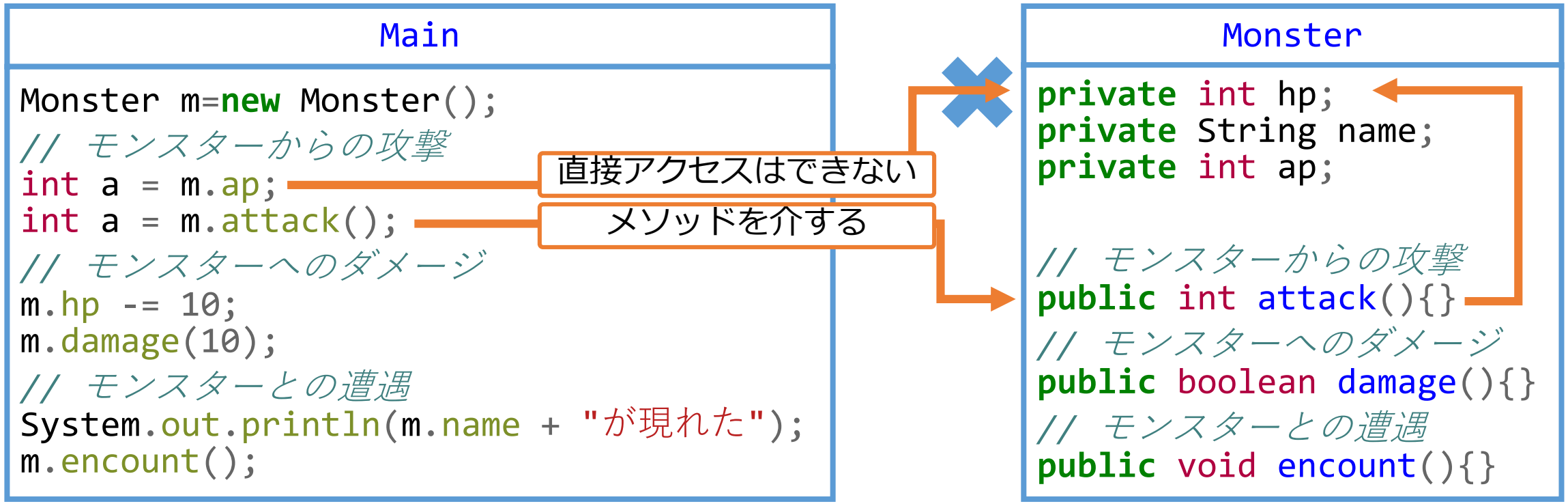
```
public class Main {
  public static void main(String[] args){
    Monster m = new Monster();
    System.out.println(m.hp);
    System.out.println(m.name);
    System.out.println(m.attack());
    System.out.println(m.getRand(10));
  }
}
```

publicや**private**を用いることで，外からのアクセスを制限できる

必要なもの以外を**private**にすることをカプセル化という．

カプセル化

- フィールドは全てprivateにし，メソッドを経由しなければフィールドにアクセスできなくすることが一般的である．



カプセル化

- オブジェクト指向ではMonsterに関する処理はMonsterに任せるといった具合に，各オブジェクトに関する処理をまとめる.

Main

```
Monster m=new Monster();
// モンスターからの攻撃
int a = m.ap;
int a = m.attack();
// モンスターへのダメージ
m.hp -= 10;
m.damage(10);
// モンスターとの遭遇
System.out.println(m.name + "が現れた");
m.encount();
```

ダメージを与える処理をmain()で書いてもよいが，0を下回る際のエラー処理や，将来的に，防御力を考慮する時もここで毎回やるの？

本来main()では，10ダメージを与え，Monsterが倒せたか否かだけが分かればよいのでは？

Monster

```
private int hp;
ng name;
ap;

からの攻撃
public int attack(){}
へのダメージ
an damage(){}

// モンスターとの遭遇
public void encount(){}
```

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．

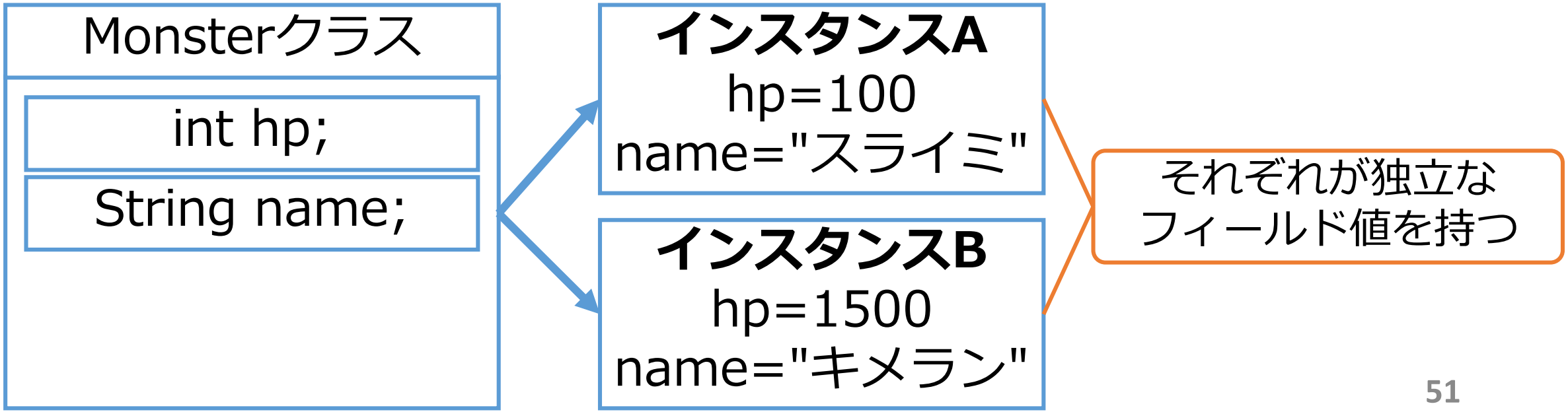
一般的に**final**定数の命名は
アンダーバー区切りの
全て大文字を使用する．

Cでいう#defineに近い．

修飾子
static

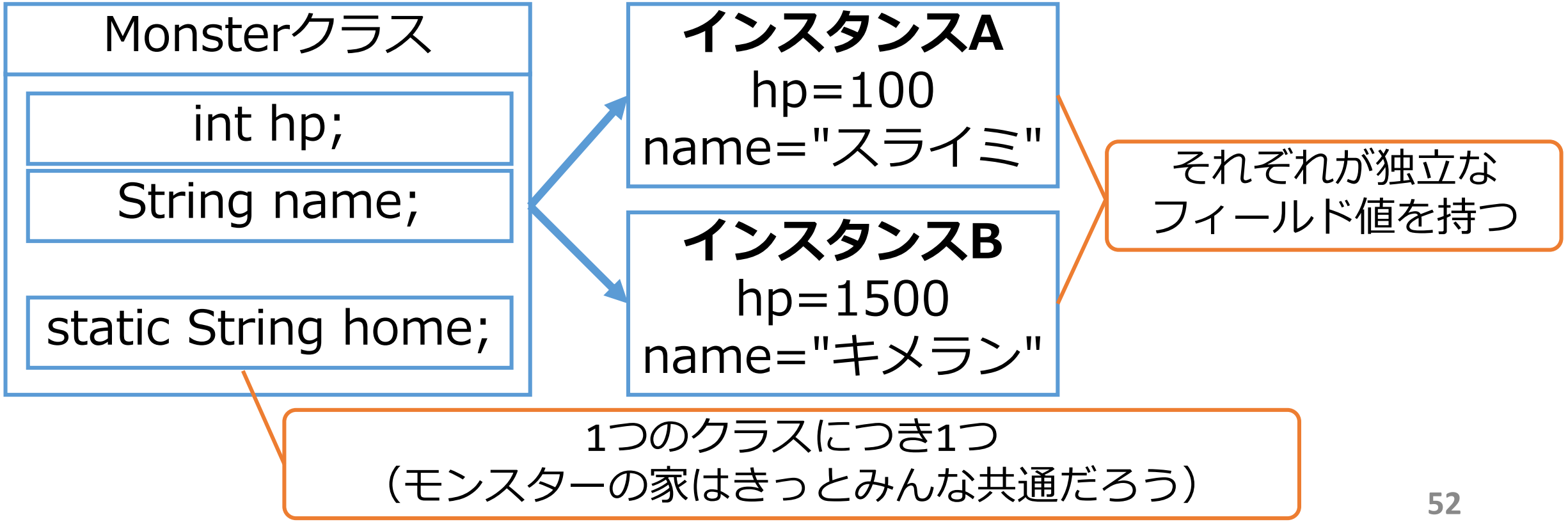
staticは，静的メンバを宣言するための修飾子である．

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

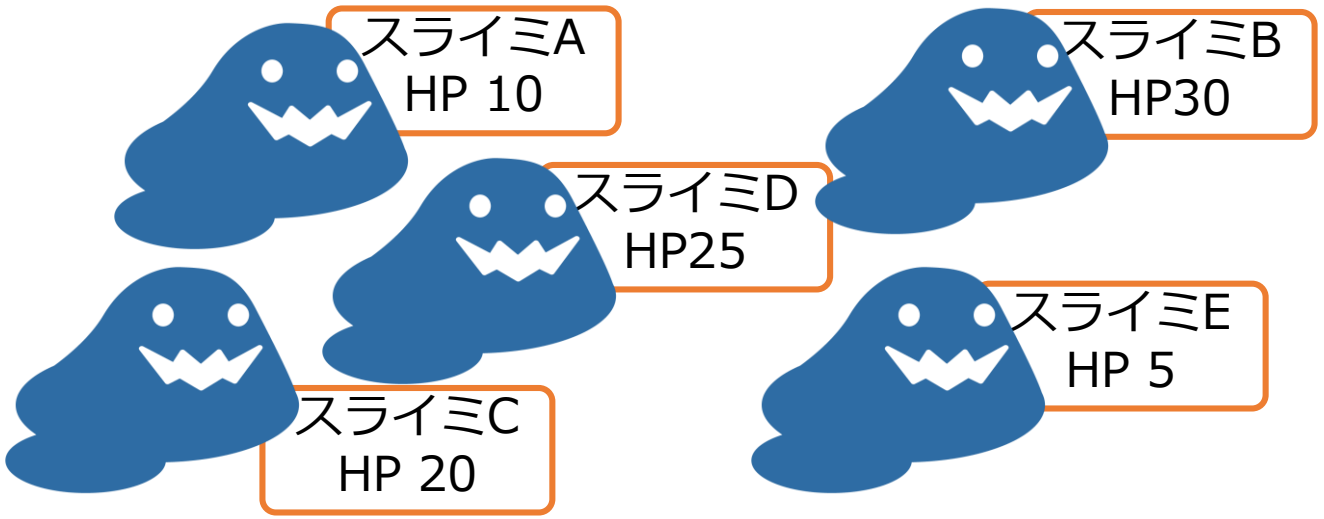
一方**static**が**ついているフィールド**は静的メンバと呼ばれ，
1つのクラスにつき1つ固有のフィールドしか持たない.



修飾子
static

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

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);
        System.out.println(Monster.home);
}
```

```
魔王の城
魔物の巣窟
魔物の巣窟
```

[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．

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メソッドは**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

次週予告

※次週以降も計算機室

前半

抽象クラス，インタフェースという新しい考え方を学ぶ.

後半

講義内容に関するプログラミング演習課題に取り組む.

演習

- 昼休み，いつものWebページに演習問題をPDFで演習問題をアップロードする．各自実施してプロII同様のWebページから提出すること.
- 質問は３人体制で受け付けるので遠慮なく申し出る．質問の際は，どこまでわかっていて何がわからないのかを申し出ること.
- （ないとは思うが）コピペは発覚次第両成敗する.
  - ✓ {コピペ，カンニング} ∈不正行為
- つまらないミスも今回は問答無用で×とするので，最終チェックを怠らないこと．（昨年は目視で甘めに採点していたが，自動採点を開発している意味がないので．．．）