

継承, カプセル化 2

2017/11/20(月) プログラミングII 第七回
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.



お知らせ 中間試験

成績評価は各レポートと中間試験を総合評価します。

中間試験は以下2種類で構成されます。

WebClassでの穴埋めテスト（10～20問？）

＞覚えておいてほしい知識を問います。

プログラム提出問題（2～3問？）

＞実際にプログラムが書けるかを問います。

＞途中までの場合はエラーのままでも提出可能です。

持ち込み不可（Web検索も不可）

WebClassテストではeclipse使用禁止で、プログラム提出課題ではeclipseで過去のソースを閲覧可能とする。

お知らせ 中間試験

試験範囲は第一回～第七回までに実施した内容すべて。

C言語に近いJavaの文法

→ 常識

Java独自の文法

→ 試験に出したくなるよね

メソッド

→ 常識

クラスとインスタンス

→ Javaで最も重要！

継承とカプセル化

→ Javaでとても重要！

皆さんにとって、教員はテスト（モンスター）を送り込んでくる魔王（黒幕）みたいなものである。相手の思考を予測すれば、魔王が出してきそうなモンスターは読めるのでは．．

お知らせ 中間試験

基本方針

WebClass 通常の講義科目の試験の要領で作問

- (大前提) 資料を読んで理解
- (得点UPに向けて) 用語の意味等を暗記
- (発展) 穴埋め問題や出力予測問題の練習

プログラム いつもの演習問題のノリ

- いつもの演習問題を余裕でこなせるように復習

前回の提出課題について

- 感想欄を書いてくれる人は減ってきましたが、書いてくれている人、ありがとうございます。
- 特にいただいているご意見が以下2点でした。
 - 練習問題の解答表示時間が短い
 - 時間外に演習課題をじっくりやりたい
→ 本日の演習課題の期限は**11月26日（日） 23 : 59**まで
- その他色々ご提案いただいているので、できる限り改善していきます。

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

前回の復習 + 質問への解答を通じて、
前回理解しきれなかった範囲の理解を深める。

目標

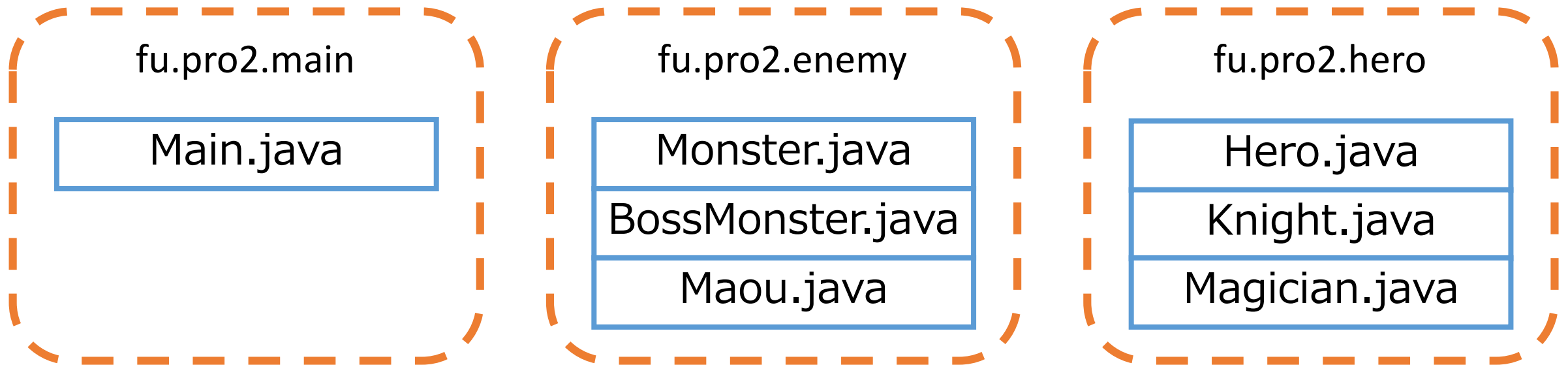
演習問題を通じて、継承とカプセル化を
実際にプログラミングできる。

本日の目次

- パッケージとimport
- 復習と質問に対する回答
- 課題演習

パッケージとimport

パッケージ：複数のクラスファイルのグループ
→フォルダみたいなもの



パッケージ名はxxx.yyy.packagenameとすることが多く、xxxやyyyには所属やアプリ名が入ることが多い。 9

パッケージとimport

パッケージ化すると、他パッケージのクラスに容易にアクセスできなくなる。

```
Monster m = new Monster(100, "スライミα", 5);
```

↓ このように書かなければ、別パッケージのクラスが使えない。

```
fu.pro2.enemy.Monster m = new fu.pro2.enemy.Monster(100, "スライミα", 5);
```

fu.pro2.main

Main.java

fu.pro2.enemy

Monster.java

BossMonster.java

Maou.java

fu.pro2.hero

Hero.java

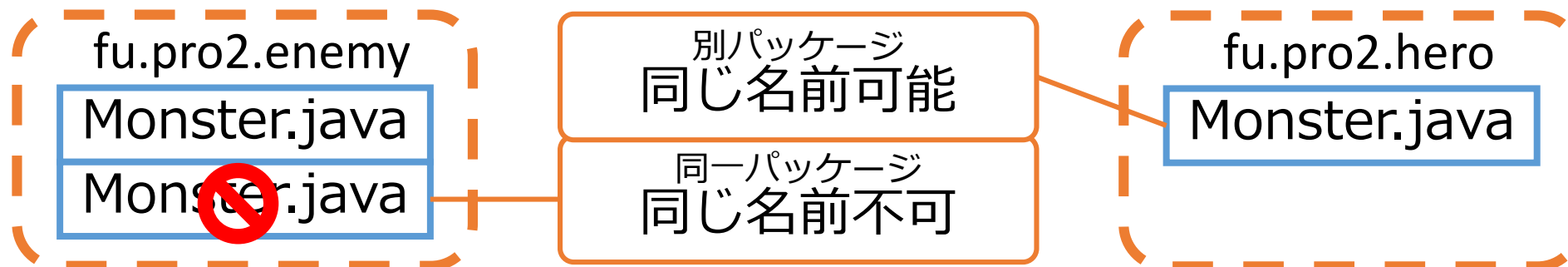
Knight.java

Magician.java

パッケージとimport

パッケージ化することの利点

- 関係のないクラスに容易にアクセスできなくなる.
 - 影響範囲の切り分けができる.
 - 予期しないクラスの使用を防止できる.
- クラスファイルの分類や整理ができる.
 - パッケージの形に添ってフォルダ分けされる.
- クラス名の衝突を防ぐことができる.
 - パッケージが異なれば、同じクラス名が使用できる.



パッケージとimport

- 自分が所属するパッケージは以下のように記述する.

```
package [所属したいパッケージ名];
```

- クラス使用時に毎回長いパッケージ名を書くのは面倒なので, import文がある. (C言語でいう#include)

```
import [パッケージ名].[クラス名];
```

```
package fu.pro2.main;  
import fu.pro2.enemy.Monster;
```

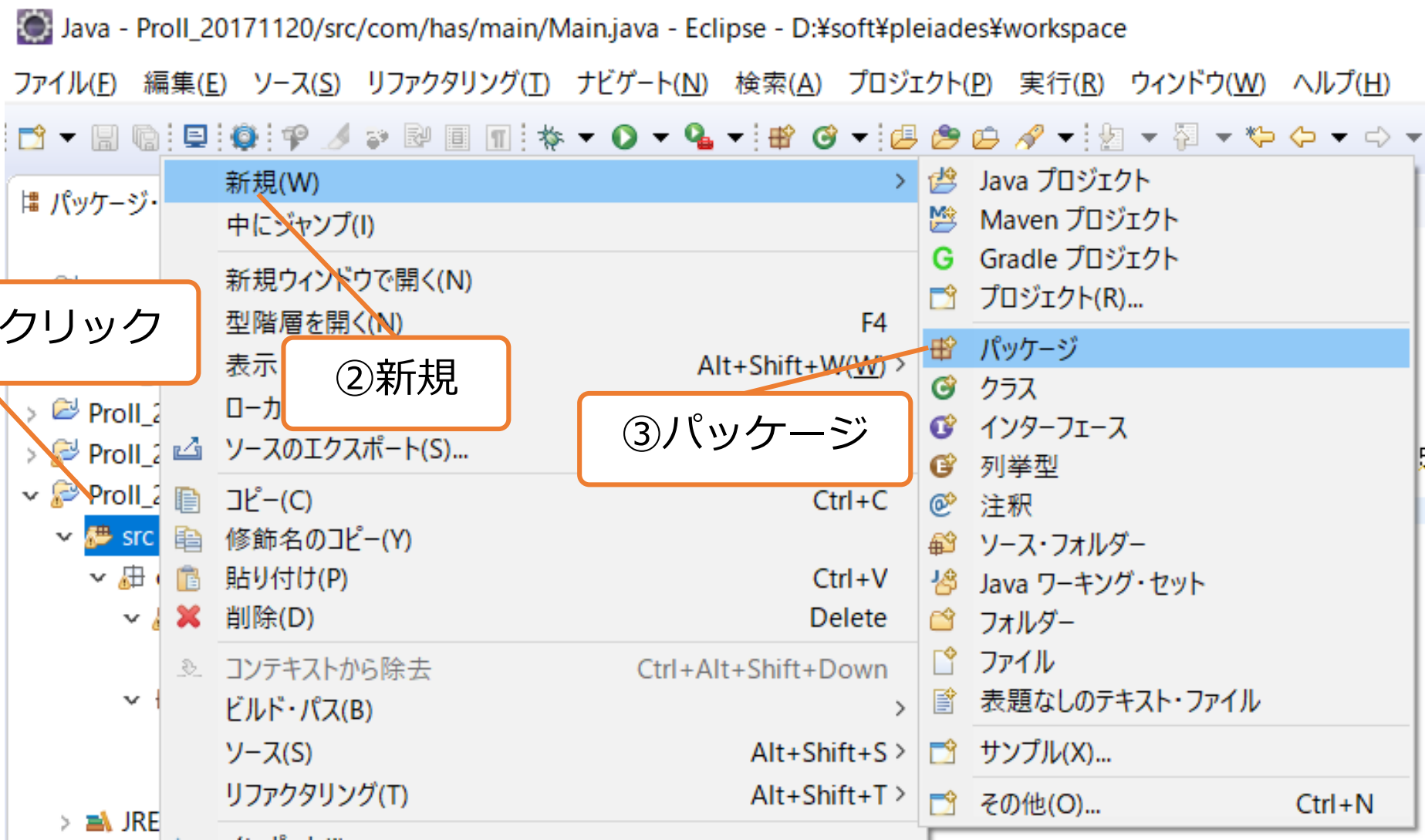
```
public class Main {  
    public static void main(String args[]){  
        Monster m = new Monster(100, "スライム", 20);  
    }  
}
```

Main.javaはfu.pro2.mainパッケージに所属

fu.pro2.enemyパッケージに所属のMonsterをimportし, 簡単に使用できるようにした.

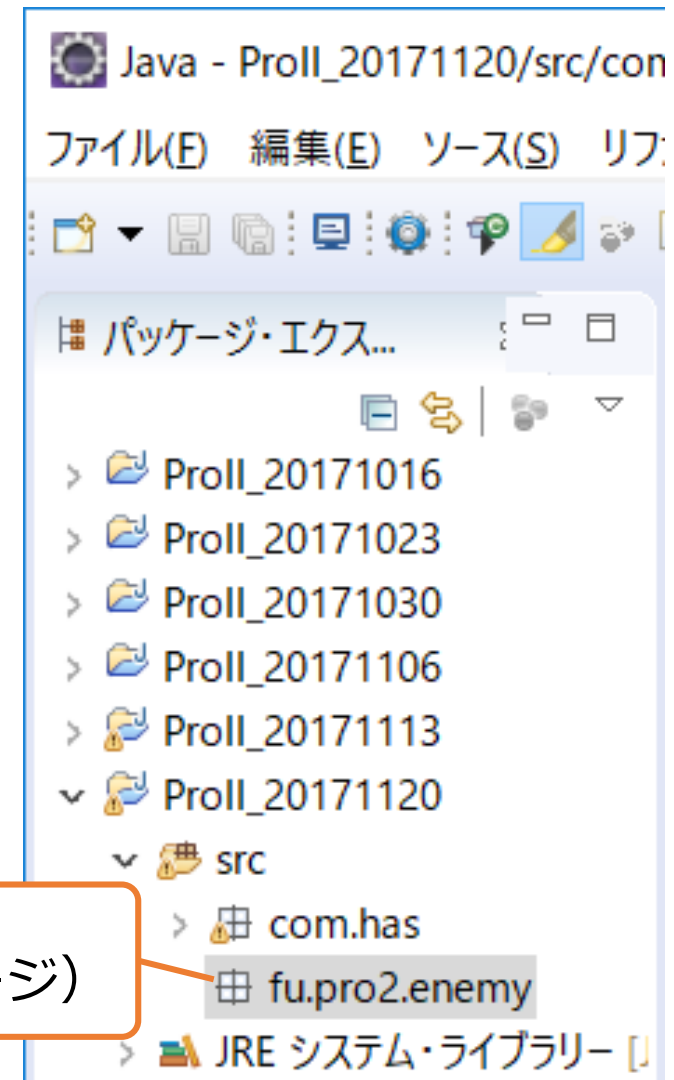
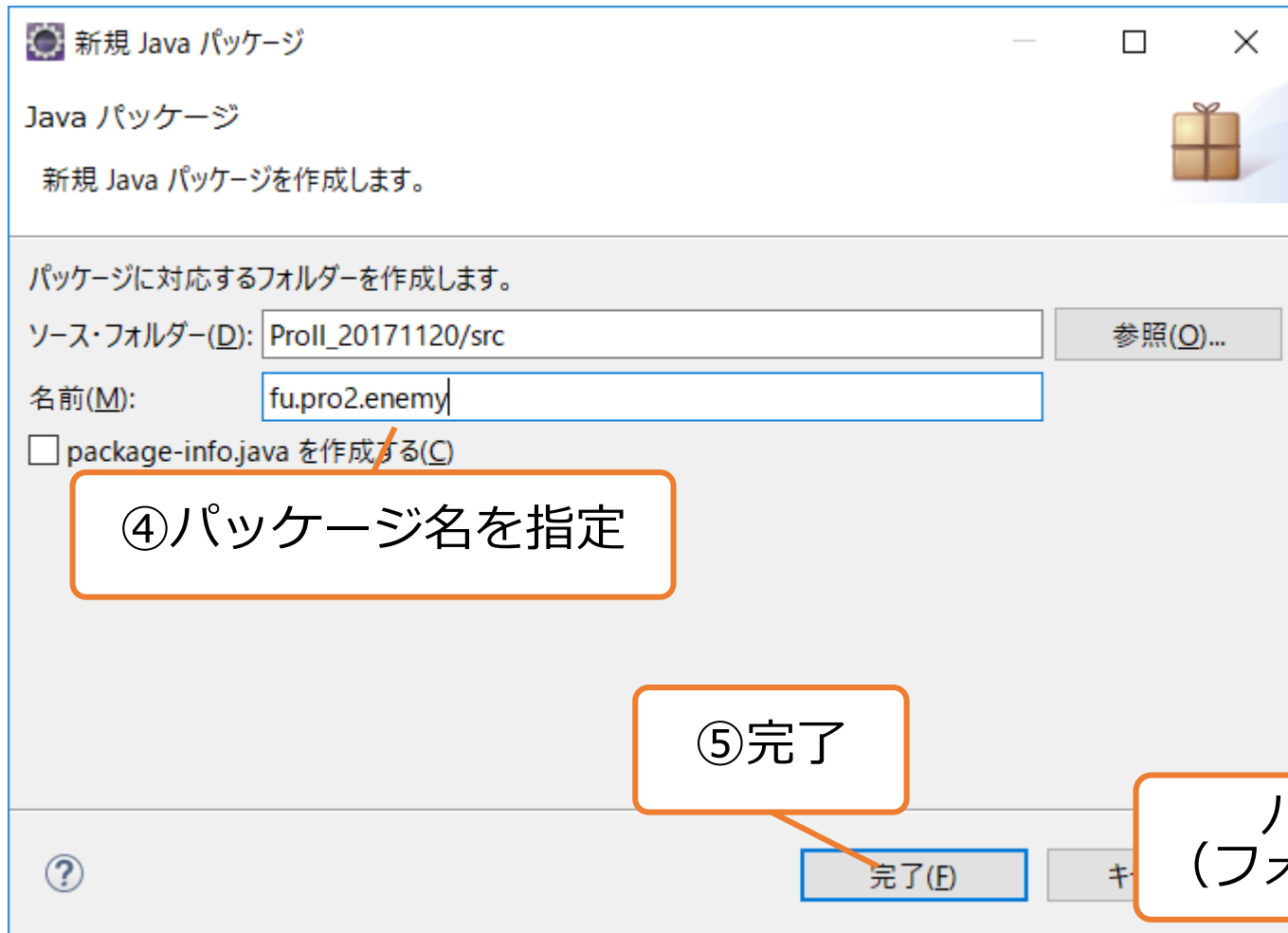
パッケージとimport

新規パッケージの作成



パッケージとimport

新規パッケージの作成

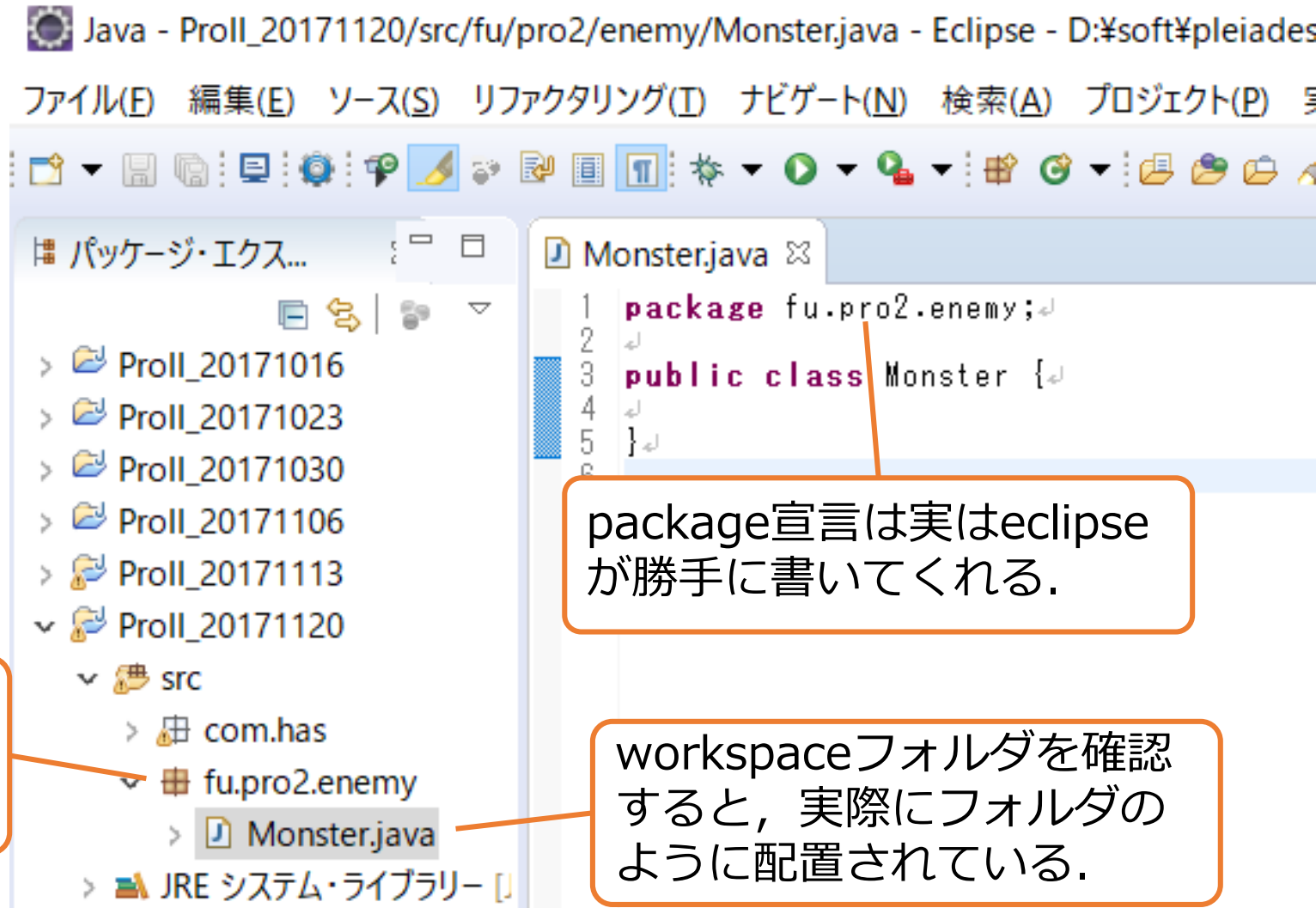


パッケージとimport

新規パッケージの作成

パッケージ（フォルダ）を作成したので、クラスファイルを作成する。

fu.pro2.enemyで右クリック→新規→クラスでMonster.javaを作成した。



パッケージとimport

練習問題 0 : 新規パッケージの作成とimport

1. 本日のプロジェクトを作成する (ProII_20171120等) .
2. srcに新規→パッケージでfu.pro2.rpgパッケージを作成する.
3. rpgパッケージにMonster.javaを作成し, いつも通りのフィールド (hp, name, ap) とattack()メソッドを定義する.
4. srcにfu.pro2.mainパッケージを作成する.
5. mainパッケージにMain.javaを作成し, Monsterインスタンスを作成する. (記述するだけではimport文がないためエラーとなるが, 保存するとeclipseが自動でimport文を記述してくれるはず.)

本日の目次

- パッケージとimport
- 復習と質問に対する回答
- 課題演習

復習の量が多いので、前回は
余裕だった人は演習課題を
やっていてくれてよい。
前回とは異なる説明を追加している
ので、参考にはなると思う。

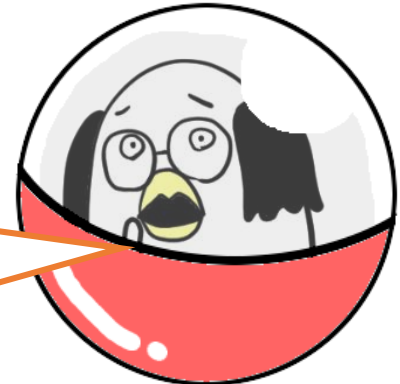


復習と質問に対する回答

カプセル化

- **カプセル化** : フィールドやメソッド, クラスに対してアクセス制御を行い, 内部を隠蔽すること.
- アクセス修飾子 : これまでおまじないとしてきたpublic等のこと.
 - **public** : すべてのクラスからアクセスできる.
 - **private** : 自クラスからのみアクセスできる.
 - **protected** : 自分を継承したクラスから
or 同じパッケージに属するクラスからアクセスできる.
 - 何も書かない : 同じパッケージに属するクラスからアクセスできる.

アクセス修飾子はこの四つだけです.



復習と質問に対する回答

カプセル化

public	: すべてのクラスから
private	: 自クラスからのみ
protected	: 自分を継承したクラス or 同じパッケージ
何もなし	: 同じパッケージ

fu.pro2.main

Main.java

HeroMonster
extends Monster

fu.pro2.enemy

Monster.java

```
public int publ;  
private int priv;  
protected int prot;  
int nothing;
```

```
void test(){  
    this.publ = 0;  
    this.priv = 0;  
    this.prot = 0;  
    this.nothing = 0;  
}
```

自クラスからは全て
アクセス可能

BossMonster
extends Monster

```
void test(){  
    super.publ = 0;  
    super.priv = 0;  
    super.prot = 0;  
    super.nothing = 0;  
}
```

同じパッケージの
クラスからは
private以外可能

復習と質問に対する回答

カプセル化

public : すべてのクラスから
private : 自クラスからのみ
protected : 自分を継承したクラス
or 同じパッケージ
何もなし : 同じパッケージ

fu.pro2.main

Main.java

```
void test(){
    Monster m =
        new Monster();
    m.publ = 0;
    m.priv = 0;
    m.prot = 0;
    m.nothing = 0;
}
```

別パッケージの
無関係クラスからは
publicのみ可能

HeroMonster
extends Monster

```
void test(){
    super.publ = 0;
    super.priv = 0;
    super.prot = 0;
    super.nothing = 0;
}
```

別パッケージの子
クラスからはpublicと
protectedのみ可能

fu.pro2.enemy

Monster.java

```
public int publ;
private int priv;
protected int prot;
int nothing;

void test(){
    this.publ = 0;
    this.priv = 0;
    this.prot = 0;
    this.nothing = 0;
}
```

BossMonster
extends Monster

復習と質問に対する回答

カプセル化

カプセル化をうまく使えるようになるには次を意識する.

フィールドに対する使い分け

- **public** : 基本フィールドには使わない.
- **private** : **非公開にしたい場合**に使う.
- **protected** : **subclasses にフィールド編集を許可する場合**に使う.
- 何も書かない : パッケージを一人で開発しているときには使っても問題はない.

メソッドに対する使い分け

- 基本フィールドと同様だが, publicも使ってよい.
- 内部処理のみを目的とするメソッドはprivateにしておく方がよい.

復習と質問に対する回答

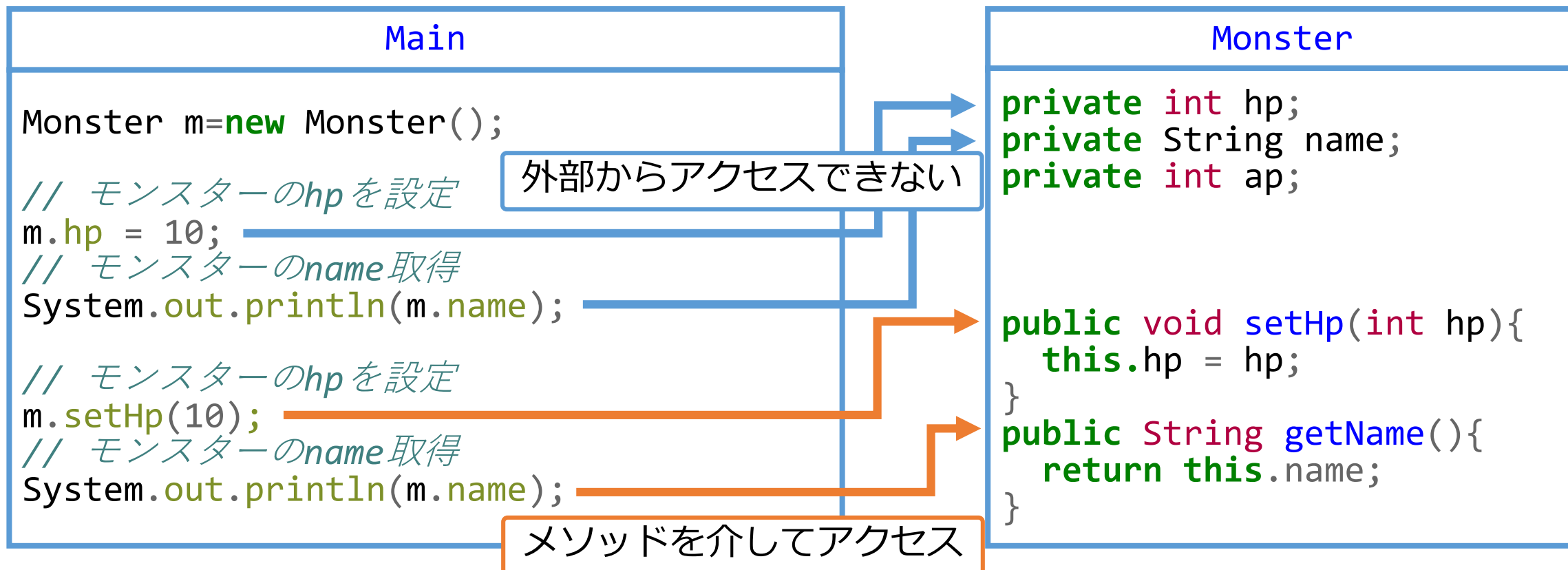
カプセル化

- privateはC言語でいうstaticと同じか？
 - その通り． 外部変数や関数に対するstaticはJavaのprivateとほとんど同じ動きをする（C言語はファイル単位でアクセス制限だが，Javaはクラス単位でアクセス制限である）．
- 今後の演習ではカプセル化は必須か？
 - 課題ごとに明示する予定である．
 - this.とsuper.の使い分けは，**違いを理解しておくこと**が重要であるため，プログラムを書く際は正直どちらでもよい．
- カプセル化をするのは手間だと思う．
 - その通り． カプセル化や継承はヒューマンエラーのリスクを減らすためのテクニックであり大規模開発で役立ってくるが，実際に使用するのは若干面倒ではある．

カプセル化

getterとsetter

フィールドを隠蔽したら、どうやってhpにアクセスするの？
> getterやsetterを使うという手がある。




カプセル化

getterとsetter

setter : 値を設定するためのメソッド
`public void setHensu(int hensu){}`

getter : 値を取得するためのメソッド
`public int getHensu(){}`



例えば、getterだけを実装することで、書き換え不可だが読み出しは可能なフィールドを実現できます。

この場合は、hpは誰でも書き換えのみ可能、nameは誰でも読み出しのみ可能となっている。

Monster

```
private int hp;  
private String name;  
private int ap;  
  
public void setHp(int hp){  
    this.hp = hp;  
}  
  
public String getName(){  
    return this.name;  
}
```


復習と質問に対する回答

getterとsetter

- フィールドへのアクセスはメソッドやコンストラクタを利用するので？ getterやsetterの必要性がわからない。
 - エラー処理などをせず、値の受け渡しだけを行うメソッドがgetter, setterである（getterやsetterもメソッドの一種）。
 - main側などで値を直接取り扱いたいときに実装するが．．．
- setterはともかくgetterをなぜ多用してはいけないのか。

完全に禁止
ではない

 - setter, getterの使用可否に関しては、**実は様々な議論があり**、教科書では普通に使っていたりするし、企業でもたぶん使っている。
 - setterを実装すると、結局フィールドをpublicにしているのと変わらなくなってしまう（誰でも改変できてしまう）。
 - getterを実装する = main等で値を直接用いることだが、それは本来クラス側が担うべき処理なのではないか？とも考えられ、それをクラスに委託するように考えるのがオブジェクト指向である。

復習と質問に対する回答

継承

継承 (extends) を使ってBossMonsterクラスを作る

is not ファイル名

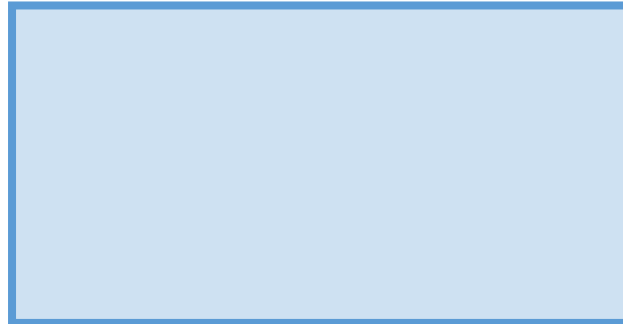
Monsterクラスがコピペされている
前提と考えることができる。

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{
```



差分のみを
記述する。

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

復習と質問に対する回答

継承

継承関係は右図のように図示する。

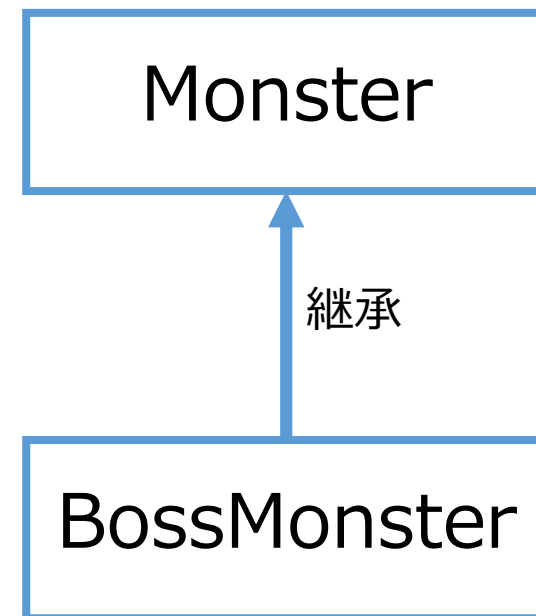
継承される側：親クラス or スーパークラス

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。

- 継承関係の矢印は逆向きの方が分かりやすいのではないかと思った。
 - 気持ちはわかる。ただ、継承には正しい継承という考えがあり、それに従うと右図のようになる。
 - ソフトウェア工学あたりで学習するクラス図では継承関係をこのような矢印で記述する。

親クラス or スーパークラス



子クラス or サブクラス

復習と質問に対する回答

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的一种である)

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

※エンジンは車的一种ではない.

親クラスorスーパークラス

Monster

子クラス is a 親クラス
なのでこの向き

継承

BossMonster

子クラスorサブクラス

親クラスorスーパークラス

Car

継承

Engine

子クラスorサブクラス

復習と質問に対する回答

継承

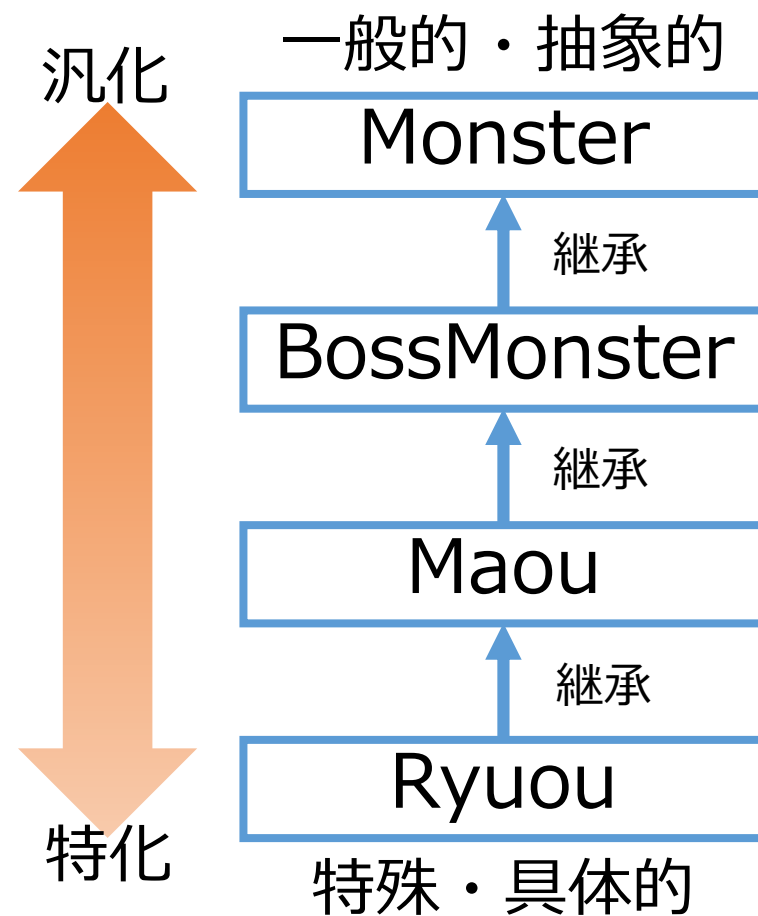
継承関係は、汎化・特化の関係ともいえる。
右の例では、

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。



復習と質問に対する回答

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.

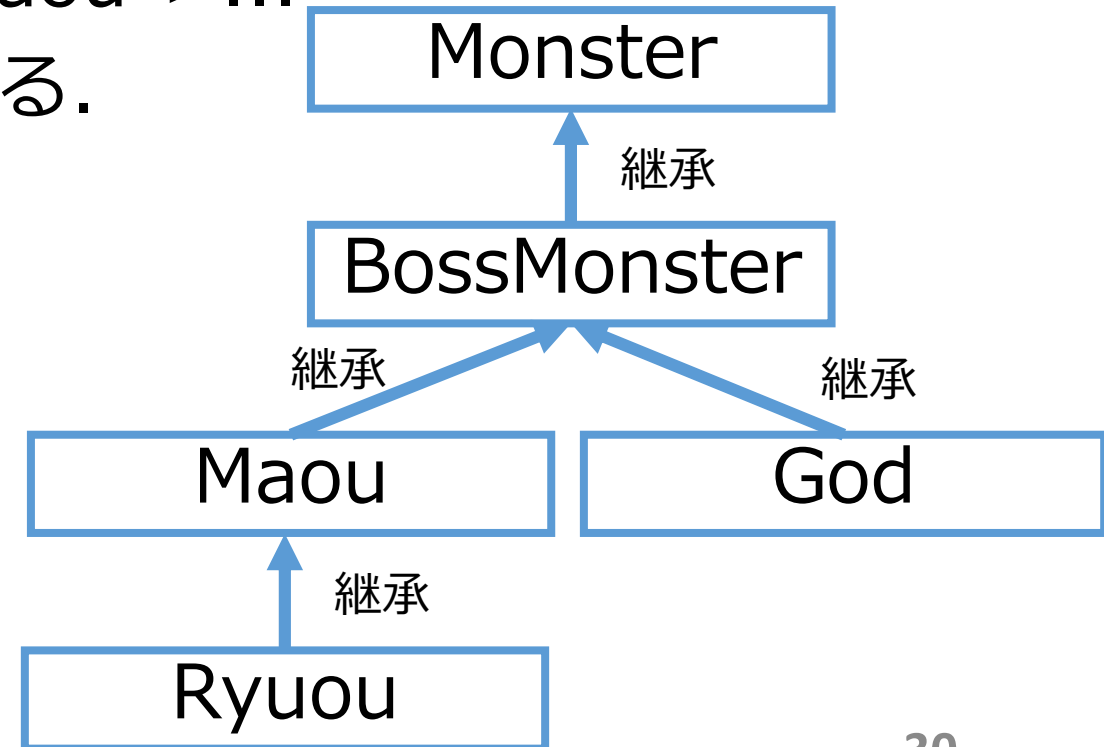
＞ Monster->BossMonster->Maou->...

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.



復習と質問に対する回答

オーバーライド

オーバーライド：親クラスと同じメソッドを**再定義**（上書き）すること

オーバーロード：自クラスに同じメソッド名を**多重定義**すること

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
全く同じ再定義
(上書き)

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

オーバーロード（多重定義）

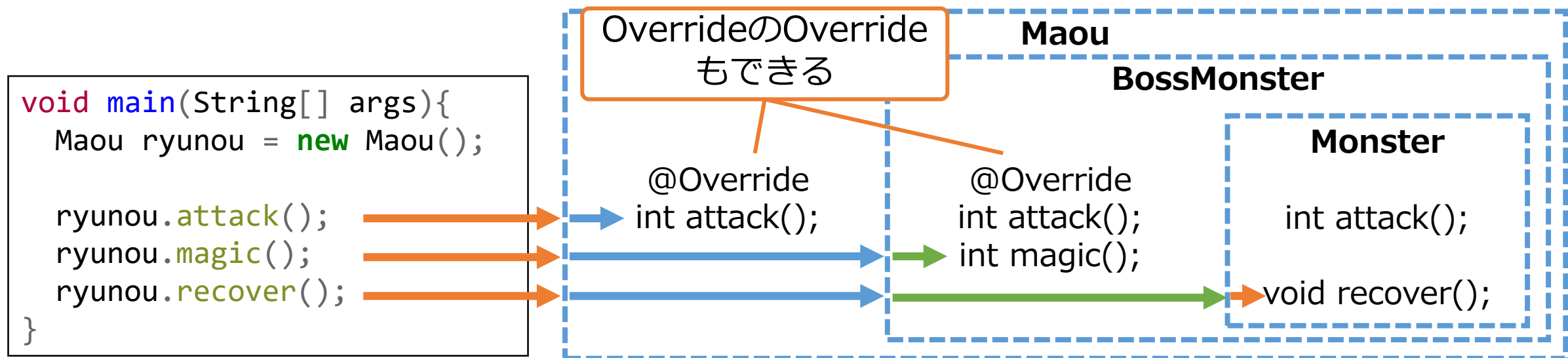
```
public int attack(int v){  
    return this.ap + v;  
}
```

同クラス内に同じ名称 + **異なる引数**のメソッドを定義すること

復習と質問に対する回答

継承, オーバーライド

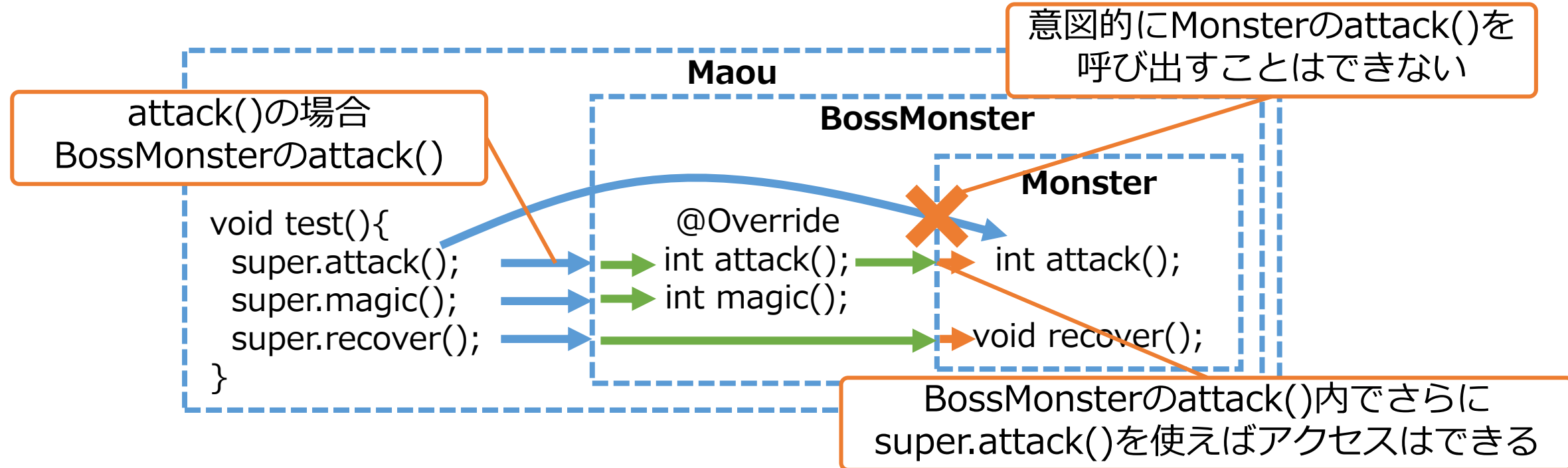
子クラスに実装されているメソッドを優先して使用する。
実装されていない場合は一つ上の親クラスに実装されている
メソッドを使用する。されていない場合は二つ上の. . .



復習と質問に対する回答

継承, オーバーライド, super.

- 親クラスのさらに親クラスに実装されているメソッドを使用する場合にはどうすればよい？（超多数の質問あり）
 - super.を使うことで、一つ上の親クラスのメソッドにアクセスできる。
 - その時呼び出されるメソッドのルールは1P前と同じ考え方をする。



復習と質問に対する回答

オーバーライド

- なぜオーバーライドを使うのか？メリットは？
 - 実は、オーバーライドが最もメリットを発揮するのは、三大要素の残り一つポリモーフィズムを考える時に判明する。
 - 例えば、「**Monster型のm**」と「**BossMonster型のbm**」をmain()内で取り扱っており、両方に攻撃させたいときを考える。オーバーライドせずBossには独自のbossAttack()というメソッドを定義した場合、m.attack();とbm.bossAttack()をmain()を書く側が意識的に使い分けなければならない。しかも、継承によりbm.attack()も実行できてしまうため、間違えてattack()と書いてしまったとしてもコンパイルエラーが出ずに気づかない（ヒューマンエラー）リスクがある。



mはMonster型だからm.attack()
bmはBossMonster型だからbm.bossAttack(). . .
面倒だからattack()でよくない！？(# °Д°)！？

復習と質問に対する回答

オーバーライド

- @Overrideの@って何？
 - 注釈を書くよ～という合図。 #includeの#みたいなもの。
 - 注釈は書いても書かなくてもよいが，書くと一定の効果を発揮する。
- @Overrideの適用範囲は？
 - この注釈の直後に書いたメソッドにのみ適用される。
- その他に@を付けることはあるのか？
 - 本講義では取り扱わないが，以下のようなものが5種類ほどある。
 - @Deprecated :そのメソッドの仕様が非推奨であることを明示的にする。 この注釈がついたメソッドやクラスを利用すると警告メッセージがコンパイル時に表示されるようになる。（下位互換のために、仕方なく残しているメソッドなどにつける等）
 - @SuppressWarnings :引数によって，コンパイル時の警告を非表示にしたりできる。

復習と質問に対する回答

super.とthis.

- super.とthis.が良くわからなかった.
 - super.は「親の」, this.は「自分の」という意味

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

```
public class BossMonster extends Monster{  
    int mp = 10;  
    public void test(){  
        this.mp += 0;  
        this.ap += 5;  
        super.ap += 5;  
        this.attack();  
        super.attack();  
    }  
    @Override  
    public int attack(){  
        super.ap += 5;  
        return super.attack();  
    }  
}
```

継承しているのでここで
`int hp = 0;`
`String name = "";`
`int ap = 0;`
が宣言されているのと同じ

フィールドはオーバーライド
しないので、基本はthis.も
super.もアクセス先は同じ

復習と質問に対する回答

super.とthis.とオーバーライド

- なぜsuper.とthis.という名前なのか.
 - 親クラスは別名スーパークラスだからsuper. を使う（恐らく） .
 - 自クラスは自分やコレという意味を込めてthis.だと思う.
- なぜメインとサブじゃなく、スーパーとサブなのか.
 - mainだとmain()メソッドと名前が競合するからでは？

復習と質問に対する回答

コンストラクタ

1. **コンストラクタ**はインスタンス生成時に一番初めに実行される処理群のことである。 **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。

```
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;  
    }  
}
```

インスタンス生成時に必ず真っ先に呼ばれる。

復習と質問に対する回答

コンストラクタ

2. コンストラクタが未定義の時, 暗黙的に引数無し
コンストラクタが存在するように振る舞う.

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

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

コンストラクタが未定義でも,
`new Monster();`が実行できる.

`public Monster(){}` が暗黙的
にここにあることになっている.

復習と質問に対する回答

コンストラクタ

3. 継承時にも**コンストラクタ**が未定義の時, 暗黙的に引数無しコンストラクタが存在するように振る舞う.

```
public static void main(String[] args){  
    BossMonster bm1 = new BossMonster();  
}
```

コンストラクタが未定義でも,
`new BossMonster();`が実行できる.

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

`public Monster(){}が暗黙的にここにあることになっている.`

```
public class BossMonster extends Monster{  
  
}
```

`public BossMonster(){}が暗黙的にここにあることになっている.`

復習と質問に対する回答

コンストラクタ

4. 継承時には**コンストラクタ**の先頭で必ず親クラスのコンストラクタを呼び出さねばならない。明示されない場合は、引数無しで暗黙的に呼び出される。

```
public static void main(String[] args){  
    BossMonster bm1 = new BossMonster();  
}
```

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

`public Monster(){}が暗黙的にここにあることになっている。`

```
public class BossMonster extends Monster{  
  
}
```

`public BossMonster(){
 super();
}`が暗黙的にここにあることになっている。


復習と質問に対する回答

コンストラクタ

5. **this()**や**super()**で自クラスのor親クラスのコンストラクタを明示的に実行することができる。ただし、コンストラクタの**先頭にしか**記述することができない。

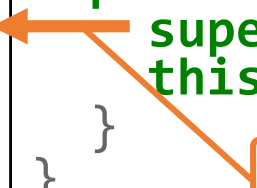
```
public static void main(String[] args){  
    BossMonster bm1 = new BossMonster();  
}
```

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



自クラスの別コンストラクタ

```
public class BossMonster extends Monster{  
    int mp = 50;  
    public BossMonster(int hp, int ap, int mp){  
        super(hp, ap);  
        this.mp = mp;  
    }  
}
```



親クラスのコンストラクタ

コンストラクタ以外（メソッド中）や先頭以外の場所に記述するとコンパイルエラーとなる。

復習と質問に対する回答

コンストラクタ

6. 親クラスに引数ありコンストラクタのみが定義されている場合、子クラスの暗黙の`super()`はエラーとなる。この時、明示的に`super(xxx)`を記述する必要がある。

```
public static void main(String[] args){  
    BossMonster bm1 = new BossMonster();  
}
```

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

Monster(){}がない！

```
public class BossMonster extends Monster{  
    int mp = 50;  
    public BossMonster(int hp, int ap, int mp){  
        this.mp = mp;  
    }  
}
```

ここに暗黙の`super()`が挿入されるが、親クラスに引数無しコンストラクタが存在しないためエラーとなる。

復習と質問に対する回答

コンストラクタ

- なぜ、必ず初めに親クラスのコンストラクタを実行しなければならないのか？
 - コンストラクタはインスタンス生成時に必ず最初に呼ばれる処理である。したがって、メソッドはコンストラクタが呼ばれている前提で処理が書かれている。
 - 子クラスは実質「親クラス + α 」である。即ちこれもコンストラクタが呼ばれている前提でメソッドが実装されている。
 - したがって、親クラスのコンストラクタは必ず最初に実行されなければならない。
- 暗黙の`super()`を挿入させないことはできるのか？
 - 親クラスのコンストラクタを何も実行しないことはできない。
 - `super(10)`等を明示することで、`super()`を挿入させないことはできる。

次週予告

※次週以降も計算機室

中間試験



本日の提出課題

課題

演習課題をEclipseで開発し,
ソースコードを提出する.

資料を公開しているサイトから, 「課題提出」で提出ページに行ける.

<http://hsgw-nas.fuis.u-fukui.ac.jp/lecture.html>

直リンクはこちら

http://hsgw-nas.fuis.u-fukui.ac.jp/lecture_file.html

演習課題

次の3ファイルを提出する。ただし、パッケージは使わない
(デフォルトパッケージを使用する) こと。

- Human.java
- Night.java
- Playboy.java

提出時にはエラーは解決しておくこと。エラーのまま提出
すると全課題の採点ができない。

感想や要望苦情等はHuman.javaのkanso()でprintln()する。

演習課題 1

Humanクラス

課題要件：

- RPGの主人公（Human）クラスを以下の条件で実装せよ.
- 数値はint型, 小数はdouble型, 文字列はString型を使用するものとする.
- フィールド
 - 名前 (name)
 - 体力 (hp)
 - 魔力 (mp)
 - レベル (lv)
- コンストラクタ
 - 3引数 (name, hp, mp)
->この時lvは1とする
 - 2引数 (name, lv)
->hpはlv*10, mpはlv*3とする

メソッド

- status() : バトル時のステータス表示
 - 引数・戻り値なし
 - 右図を出力する
- punch() : 殴る攻撃
 - 引数なし
 - 戻り値：与えるダメージ（10固定）
- item() : 道具を使う
 - 引数・戻り値なし
 - 「道具を持っていない」と表示する.
※実装していないがしてる風に見せるテク

たつひとLv(10)
HP(950)
MP(350)

名前以外
は半角

注意：フィールドはカプセル化せよ。ただし、アクセス修飾子の種類によっては、課題2で継承したときに編集できなくなることには注意せよ。

演習課題 2

職業クラス

課題要件：

- ドラ○エ 6 のダー○神殿での転職をイメージしてほしい（知らなくてもよい）．主人公（Human）は様々な職業に転職できる．職業戦士（Night）と遊び人（Playboy）クラスをHumanクラスを継承し，以下の要件を満たすクラスを開発せよ．

戦士（Night）

- コンストラクタ
 - 親クラスと同じ2種類とする．
- 独自フィールド
 - メソッドで必要ならば作ってもよい．
- 独自メソッド
 - charge() : 気合ため
 - 引数・戻り値なし
 - 次の攻撃のダメージ3倍する．
 - 「気合をためた」と表示する．
- オーバーライド
 - punch() : 親クラスのpunch()の1.5倍（切捨て）のダメージを与える．

遊び人（Playboy）

- コンストラクタ
 - 4 引数（name, hp, mp, shout）
->Lvは必ず1とする．
- 独自フィールド
 - 掛け声（shout）
- 独自メソッド
 - なし
- オーバーライド
 - punch() : 「何か叫んでいる」，「（掛け声）」の2行を表示する．
戻り値は0とする．

演習課題 3

遊び人の進化

課題要件：

- 遊び人を極めると、いくつか派生の職業に就くことができる。これらは遊び人と同等の要素を持ちつつ、独自の技などを持っている。遊び人クラスを継承し、以下の要件を満たすクラスを開発せよ。

ギャンブラー (Gambler)

- コンストラクタ
 - 親クラスと同じ1種類とする。
- 独自フィールド&メソッド
 - 必要ならば作ってもよい。
- オーバーライド
 - `punch()` : 必ず3回に1回100のダメージを与え、自身のmpを10消費する。mpが足りない場合や、それ以外の場合は、親クラスと同じ挙動をする。初回実行を1回目とし3回目、6回目、 , , が100となる。

ニート (Neet)

- コンストラクタ
 - 3引数 (name, hp, mp)
 - >Lvは必ず1とする。
 - >shoutは「・・・」とする。
- 独自フィールド&メソッド
 - なし
- オーバーライド
 - `punch()` : 自身のhpが3以下の時、500のダメージを与える。それ以外は親クラスと同じ挙動をする。