抽象クラス・インタフェース

2018/4/24(火)　プログラミングIV 第三回

福井大学　工学研究科　情報・メディア工学専攻

長谷川達人

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.

- 学籍番号を入力すると，成績を表示する仕組みを開発してみましたので使ってみてください.

質問に対する回答

継承について

- Is-aの例がよくわからない（課題４の回答を紹介）

  良い例：親：道具　　子：ハンマー→　ハンマー is a 道具

  　　親：携帯　　子：iPhone　→　iPhone is a 携帯

  悪い例：親：自転車　子：タイヤ　→　自転車 is not a タイヤ

  　　父：悟空　　子：悟飯　　→　そういう話ではない.

- 継承したときの子のメモリ領域はどうなっているのか？
  - インスタンス化時に親クラスも含めた領域が確保されるだけ.
- 継承したい複数クラスの要素全てを持ったクラスを作成し，継承すれば複数クラスの継承が実現できるのではないか.
  - 元の複数クラス→全て持ったクラス　がつながらないので，これは継承ではない.

質問に対する回答
オーバーライドについて

- オーバーライドする対象は親クラスのみか？
  - 親クラスのみ
- 親クラスのものを呼び出す時，なぜsuperなのか？
  - 親クラスは別名スーパークラスなので.
- オーバーライド時の親クラスのメソッドの呼び出し方がわからない.
  - super([引数]) : 親クラスのコンストラクタ
  - super.method([引数]) : 親クラスのmethod()
  - this([引数]) : 自クラスのコンストラクタ
  - this.method([引数]) : 自クラスのmethod()

質問に対する回答

色々

- 親<-子<-孫クラスと継承した時，孫から親のメソッド等にアクセスできるのか.
  - できない．親クラスのメソッド等をsuper.method()でアクセスできるので，親クラスに仲介用メソッドを定義し，その中でsuper.method()とすることで間接的にはアクセス可能である.
- プログラムが長くなったとき，可読性を上げるには？
  - まず，長くならないようにシンプルで賢い実装を心がける.
  - メソッド数が増える場合，各メソッドが何をするメソッドなのか，わかりやすい命名をする.
  - getterやsetterを同じゾーンにまとめておく，似た処理を行うメソッドは同じゾーンにまとめておく.

質問に対する回答

色々

- ソースコードを横に並べる方法を教えて.
  - ファイル名のところをドラッグする.
- 変数の初期化は必ず必要か？
  - 前回の質疑で紹介した初期化の値はフィールドには適応されるが，ローカル変数の場合は初期化しないと参照できない.
- 今後の授業で難しいところはあるか？
  - ポリモーフィズム（次回），ジェネリクス（第８回）
- String args[]とString[] argsとString argsの違いは？
  - 前２つはどちらも同じ，最後のやつは配列ではない.

質問に対する回答
カプセル化

- カプセル化の必要性がよくわからない
  - オブジェクトに関連する処理はオブジェクト側が担当することで，ヒューマンエラーのリスクを下げるため.

```java
public class Monster {
  private int hp = 0;

  public Monster(int hp){
    this.hp = hp;
  }

  // モンスターにダメージを与える
  public void damage(int damage){
    this.hp -= damage;
    if(this.hp < 0){
      this.hp = 0;
      System.out.println("倒した！");
    }
  }
}
```

```java
public class Main {
  public static void main(String[] args){
    Monster m = new Monster(10);

    // 直接フィールドを操作できる，
    // hpが0以下の時の処理をしない人がいるかも？
    m.hp -= 100;

    // メソッドを介することで，フィールド操作に
    // 関してはMonster側が一貫して担当できる.
    m.damage(100);
  }
}
```

質問に対する回答

課題と今後について

- 課題XXの○○は～～な手法でもよいのか？
  - 基本的には課題の要求を実装できていれば手段は問わない.
  - 練習や演習はその日やった内容の理解を補足することを目的としているので，PPTの内容を理解することが重要である.
- super()はテストでは明示しなければならないのか？
  - コンパイルエラーが出ない状況ならば不要である.
- 期末は筆記試験か？
  - 未定だが，プロII同様，WebClass＋eclipse開発の予定.
- 過去の自由開発の例は？どこまで自由？
  - 今年からの試みである．どこまでも自由．既存の作り直しも可能（ただし点数は下がる）．グループではやらない予定.

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

ポリモーフィズムを学ぶ前準備として，抽象クラスと
インタフェースという概念について学ぶ.

目標

クラス，インスタンス，抽象クラス，
インタフェースの違いが説明できる.

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.

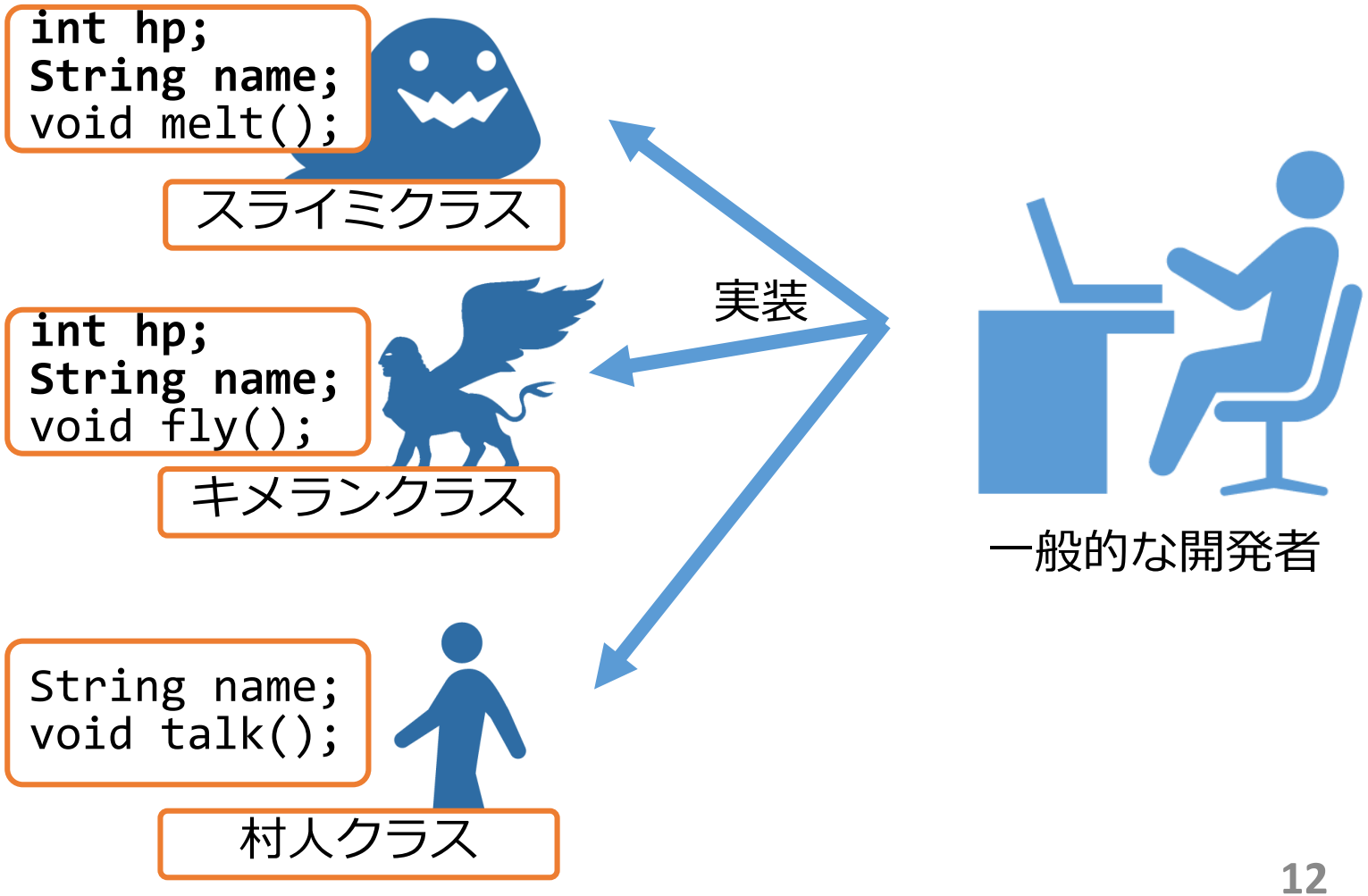
課題 2

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.

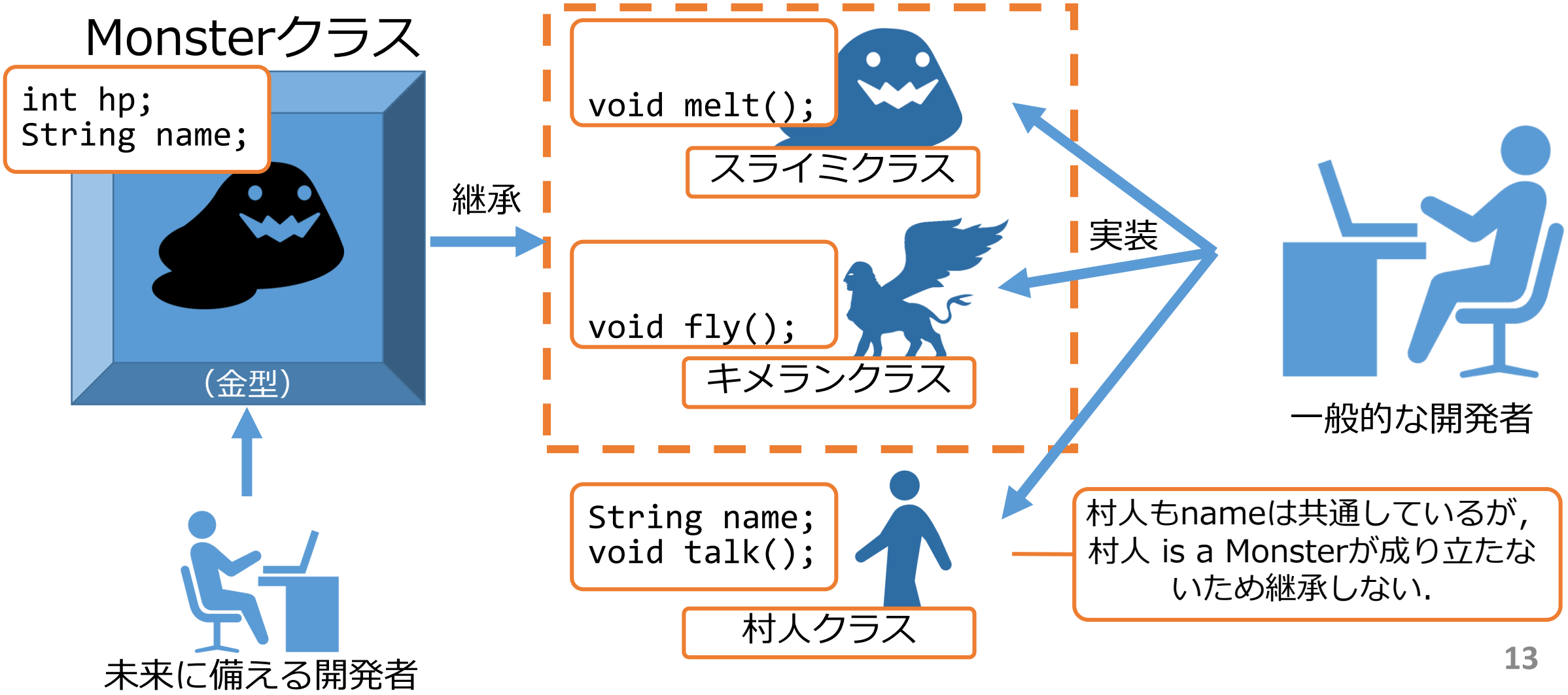
課題 3

感想（あれば）

これまで


int hp;
String name;
void melt();
スライミクラス
int hp;
String name;
void fly();
キメランクラス
String name;
void talk();
村人クラス
実装
一般的な開発者

これまで
継承元を作成する



継承の不都合

1. 継承元（親クラス）の開発時には確定できない「詳細未定メソッド」が存在しうる.
2. newでインスタンス化される可能性がある.

継承の不都合
詳細未定メソッド

1. 継承元（親クラス）の開発時には確定できない「詳細未定メソッド」が存在しうる.

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

    // 遭遇したときの処理
    public void encounter(){
        System.out.println(this.name + "があらわれた！");
    }
}
```

スライミのときはこれでいいが，
キメランのときは先制攻撃を
仕掛けてくるようにしたい.

遭遇時の処理はこれ以外の様々なパターンが起こり得る
→詳細未定メソッド

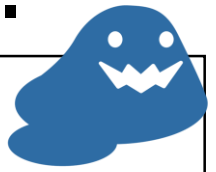
継承の不都合
詳細未定メソッド

解決案１
Monsterクラス側にencounter()を定義しない.

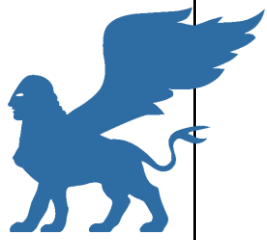
```
public class Monster {
    protected int hp = 0;
    protected String name = "";
}
```

protected修飾子は
「同じパッケージ内 or 自分を継承した子クラスからアクセス可能」

```
public class Slimi extends Monster{
  // 遭遇したときの処理
  public void encounter(){
    System.out.println(this.name + "があらわれた！");
  }
}
```



```
public class Chimeran extends Monster{
  //遭遇したときの処理
  public void encounter(){
    System.out.println(this.name + "の攻撃");
    this.attack();
  }
}
```



→継承先が実装し忘れるとencounter()が使える
として開発していた人がいた場合エラーになる.

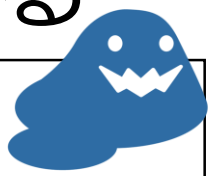
継承の不都合
詳細未定メソッド

解決案２
Monsterクラス側に空のencounter()を定義する

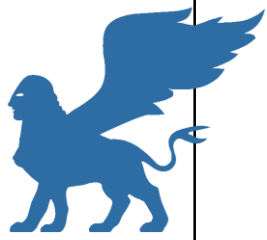
```
public class Monster {
    protected int hp = 0;
    protected String name = "";

    public void encounter(){
        //中身はオーバーライド先に
    }
}
```

```
public class Slimi extends Monster{
  @Override
  public void encounter(){
    System.out.println(this.name + "があらわれた！");
  }
}
```



```
public class Chimeran extends Monster{
  @Override
  public void encounter(){
    System.out.println(this.name + "の攻撃");
    this.attack();
  }
}
```



→継承先がencounter()を実装し忘れるリスク
→何もしないメソッドと区別がつかない

継承の不都合

newでインスタンス化される可能性がある

Monsterクラスは抽象的なクラスであって，本来インスタンス化されることを想定していない.
　→今のままではできてしまう.

解決案１
　「// newしないでね」とコメントに書く.
　→ヒューマンエラーのリスクになる.

そもそも，Monsterのように詳細未定が残る抽象的なクラスはnewできないようにするべきである.

継承の不都合

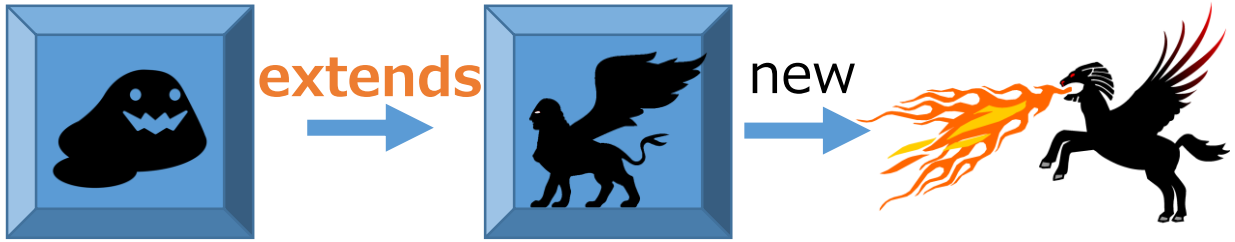
newでインスタンス化される可能性がある

クラスの使われ方は２種類ある

インスタンス化して使う

キメランクラス　　インスタンス

継承の親クラスとして使う



Monsterクラス　　キメランクラス　　インスタンス

自由に選べる２種類の利用法があることが，逆にnewされたくないのにされてしまう原因になっている.

抽象クラス

これらの不都合を解決する手段として**抽象クラス**という概念がある．宣言方法は，

1．詳細未定メソッドにabstractをつける．
2．クラス宣言時にabstractをつける．

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

    public abstract void encounter();
}
```

abstractをつける

メソッドの中身は定義しないで，
{}を書かずにセミコロンとする

抽象クラス

抽象クラスは
・newでインスタンス化できない
・抽象メソッドをオーバーライドする必要がある

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

    public abstract void encounter();
}
```

new

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

型 Monster のインスタンスを生成できません
フォーカスするには 'F2' を押下

extends

```
public class Slimi extends Monster{
    //Overrideしないと．．．
}
```

型 Slimi は継承された抽象メソッド Monster.encounter() を実装する必要があります
2 個のクイック・フィックスが使用可能です:
実装されていないメソッドの追加
型 'Slimi' を abstract にします

抽象クラス

親クラスの開発者が**抽象クラス**を用いることで

- 不意にnewされること防止することができる.
- 抽象メソッドを必ずオーバーライドさせられる.
- 「何もしないメソッド」と「詳細未定メソッド」を区別することができる.

不都合を解消です

多段階の抽象継承構造

abstract
Monster
String name;
int hp;
attack()
abstract(未定)
run()
abstract(未定)

extends

abstract
WalkingMonster
String name;
int hp;
attack()
abstract(未定)
run()
走って逃げる

extends

abstract
FlyingMonster
String name;
int hp;
attack()
abstract(未定)
run()
飛んで逃げる

extends

Draculan
String name;
int hp;
attack()
吸血パンチ
run()
走って逃げる

extends

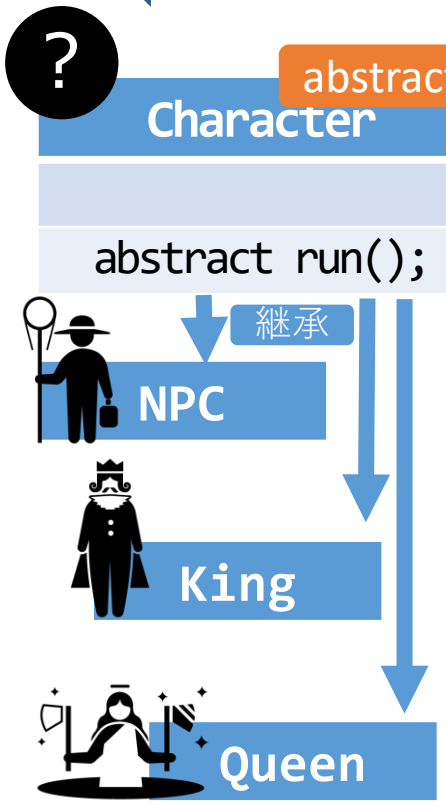
Majokkou
String name;
int hp;
attack()
箒を投棄
run()
走って逃げる

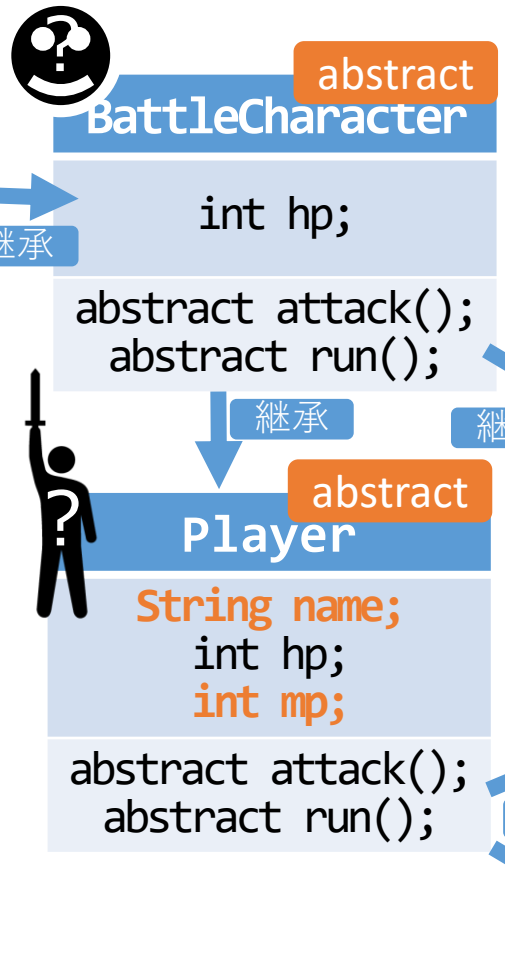
詳細未定メソッドが
残ってる限り抽象クラス

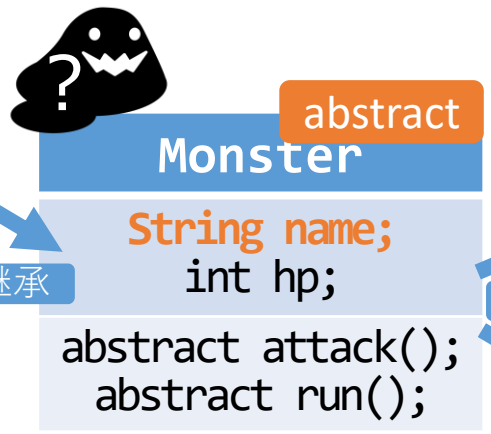
抽象クラス

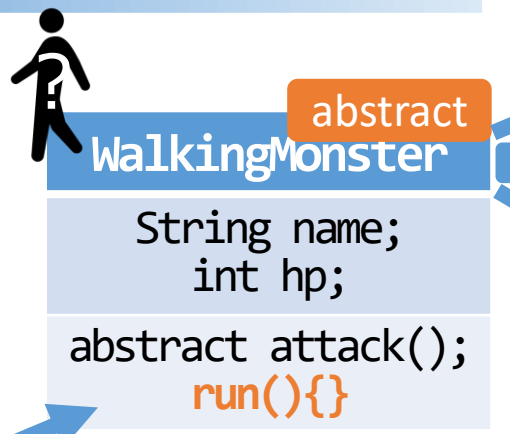
Monsterクラスをもっと抽象化していこう

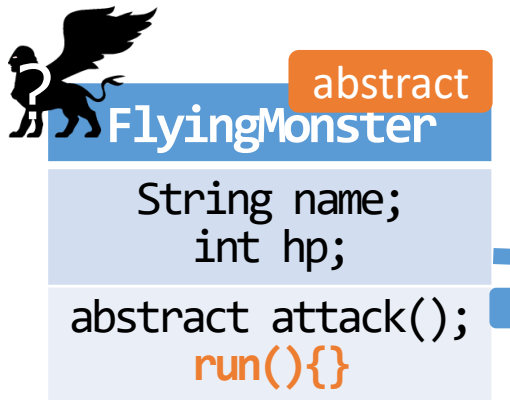
抽象的　になるほど，フィールドやメソッドが減っていく

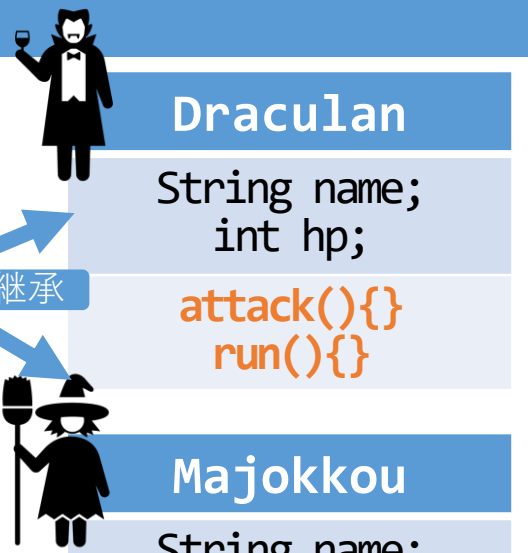


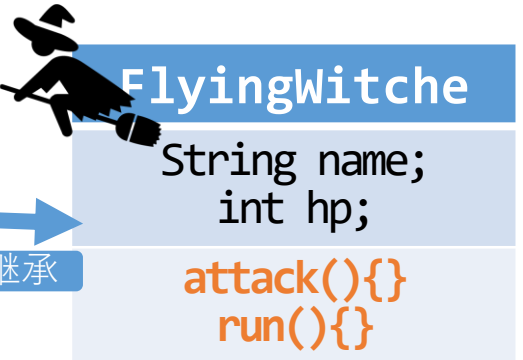






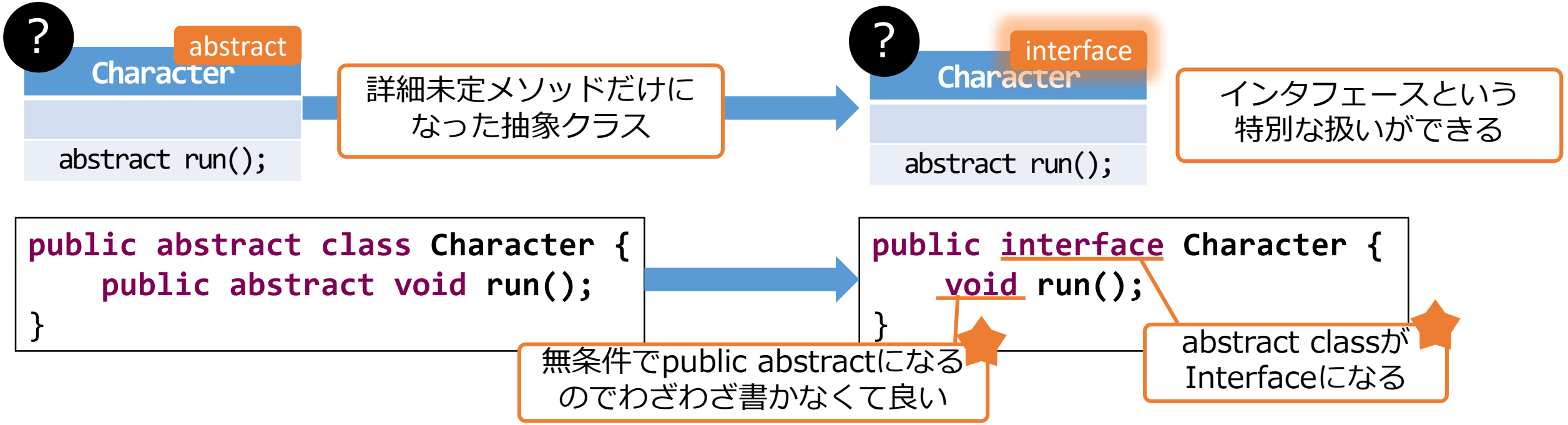






抽象クラス

Monsterクラスをもっと抽象化していこう



インタフェースにできる条件

・全てのメソッドが抽象メソッドである.

・（基本的に）フィールドを1つも持たない.

インタフェース

インタフェースはメソッドの存在のみを定義した設計書のようなものである.

インタフェースを実装するには**implements**を用いてクラスの定義を行う.

```
public class King implements Character{
    @Override
    public void run() {
        System.out.println("王様は逃げません. ");
    }
}
```

extendsではなく
implements

抽象メソッドをオーバーライドする
または，Kingをabstractクラスにする.

インタフェース
インタフェースの利点

- 複数の子クラスに同じメソッドを実装させることができる（抽象クラスと同様）.
- インタフェースがimplementsされているということは，**インタフェースに定義されているメソッドを必ず持っている**ことが保証される.
- インタフェースは**複数同時にimplementsできる**.

```
public class King implements Character, ItemUser, NPC{
    @Override
    public void run() {
        System.out.println("王様は逃げません. ");
    }
}
```

implementsは複数可能
(extendsは1つまでだった)

インタフェース

なぜ多重継承が許容されているのか

abstract

WalkingMonster

String name;
int hp;

attack()
abstract(未定)

run()
走って逃げる

abstract

FlyingMonster

String name;
int hp;

attack()
abstract(未定)

run()
飛んで逃げる

多重継承

同じメソッド名を持つクラスを多重継承した際，Overrideを忘れると，どっちを実行するかわからなくなる.

FlyingWitche

String name;
int hp;

attack()
箒ではたく

run()
どっち！！？

FlyingWitche

String name;
int hp;

attack()
箒ではたく

run()
箒で飛び逃げる

多重継承

同じメソッド名を持つinterfaceを多重継承しても，必ずOverrideしなければならないため混乱は起きない.

interface

WalkingMonster

attack()
abstract(未定)

run()
abstract(未定)

interface

FlyingMonster

attack()
abstract(未定)

run()
abstract(未定)

インタフェース

extendsとimplementsを併用する

abstract

Player

String name;
int hp;
int mp;

abstract attack();
abstract run();

interface

ItemUser

abstract useItem();
abstract checkItem();

```
public interface ItemUser {
    void userItem(int item);
    String checkItem();
}
```

extends

implements

Night

String name;
int hp;
int mp;

attack(){}
run(){}
useItem(){}
checkItem(){}

```
public class Night extends Player implements ItemUser{
    @Override
    public void attack() {}
    @Override
    public void run() {}
    @Override
    public void userItem(int item) {}
    @Override
    public String checkItem() {
        // 本来は全てのメソッドの処理を実装する.
        return null;
    }
}
```

Playerの抽象メソッドの実装

ItemUserの抽象メソッドの実装

抽象クラス

練習問題１：試しに作ってみる

1. 抽象クラスMonster作成せよ．ただし，Monsterを継承した子クラスからフィールドにアクセスできるようにすること．コンストラクタからフィールドを全て初期化すること．

   フィールド：name(文字列)，hp(数値)，mp(数値)

   メソッド

   - String getName()：nameを返す
   - void attack()：詳細未定メソッド

2. Main.javaにmainメソッドを作成し，Monsterインスタンスを作成できないことを確認せよ．

インタフェース

練習問題２：試しに作ってみる

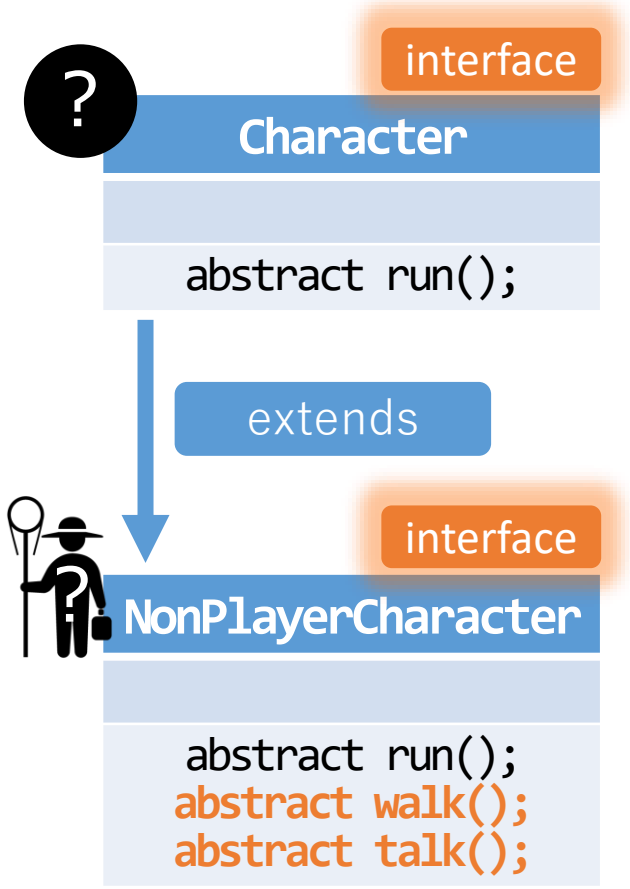
1. Magicableインタフェースを作成せよ.
   （魔法が使えるということでMagic+able）
   ・void magic()：詳細未定メソッド
2. Monsterを継承し，Magicableを実装し，魔法が使えるモンスターとして，Wizardnクラスを作成せよ．コンストラクタは親クラスと同等のものを実装せよ.
   attack()：「○○の攻撃！」と表示する.
   magic()：「○○は魔法を唱えた！」と表示し，mpを５減らす．減らせない場合はmpは減らず更に「が，失敗した」と表示する.　※○○には自身の名前が入る.
3. mainメソッドでWizardnをインスタンス化し，getName()，attack()，magic()をテストせよ．magic()はmpが切れるまで試すこと.

少し発展的な話

理解しておくとなお良い

インタフェース

1．インタフェースを継承したインタフェース




```
public interface Character {
    void run();
}
```

interfaceを作るときはinterfaceを
~~implements~~ extendsする．(ややこしい)

```
public interface NonPlayerCharacter extends Character{
    // void run(); はCharacterから継承されている
    void walk();
    String talk();
}
```

インタフェース

2．インタフェースにフィールドをもたせられる？

インタフェースにできる条件
・全てのメソッドが抽象メソッドである.
・（基本的に）フィールドを1つも持たない.

初期化済みであればフィールドを持つことが可能
ただし`public static final`が暗黙的に宣言される

暗黙的に public static final がある

暗黙的に public abstract がある

```
public interface Character {
    String VERSION = "v1,02";
    void run();
}
```

インタフェース

3．インタフェースに実装済みメソッドをもたせられる？

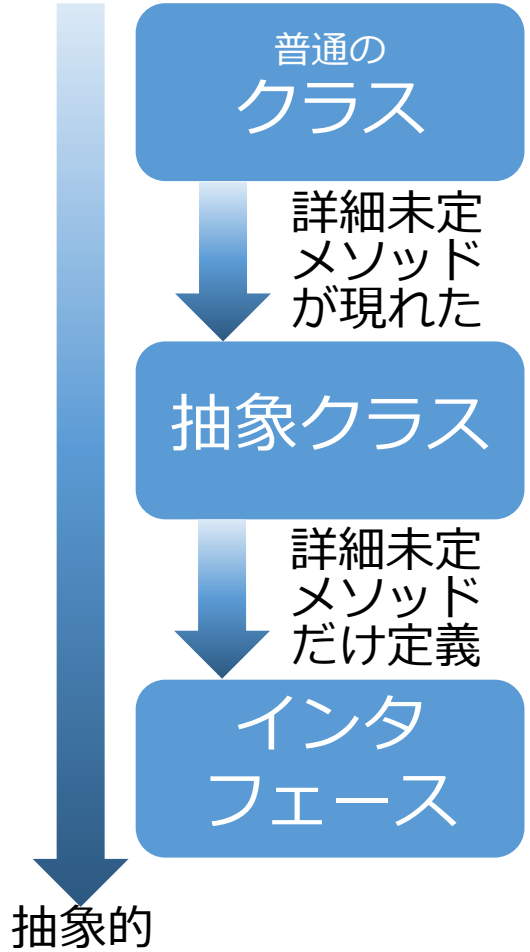
インタフェースにできる条件
・全てのメソッドが抽象メソッドである.
・（基本的に）フィールドを1つも持たない.

Java8 (2014リリース)以降はdefaultを用いることで，デフォルト処理を実装した抽象メソッドを定義できる.

```
public interface Character {
    default void run(){
        System.out.println("逃げた. ");
    }
}
```

Overrideされなかったときにデフォルトで呼び出される処理を記述できる.
ただし，多重継承による衝突には注意して利用する必要がある.

まとめ



- 各開発者に共通する部分（親クラス）を開発する立場に関する考え方を学んだ.
- newされると困る詳細未定メソッドを持つクラスを**抽象クラス**（abstract class）として定義する.
- 抽象メソッドだけとなった抽象クラスは特別に**インタフェース**（interface）として定義できる.

インタフェース
コールバック

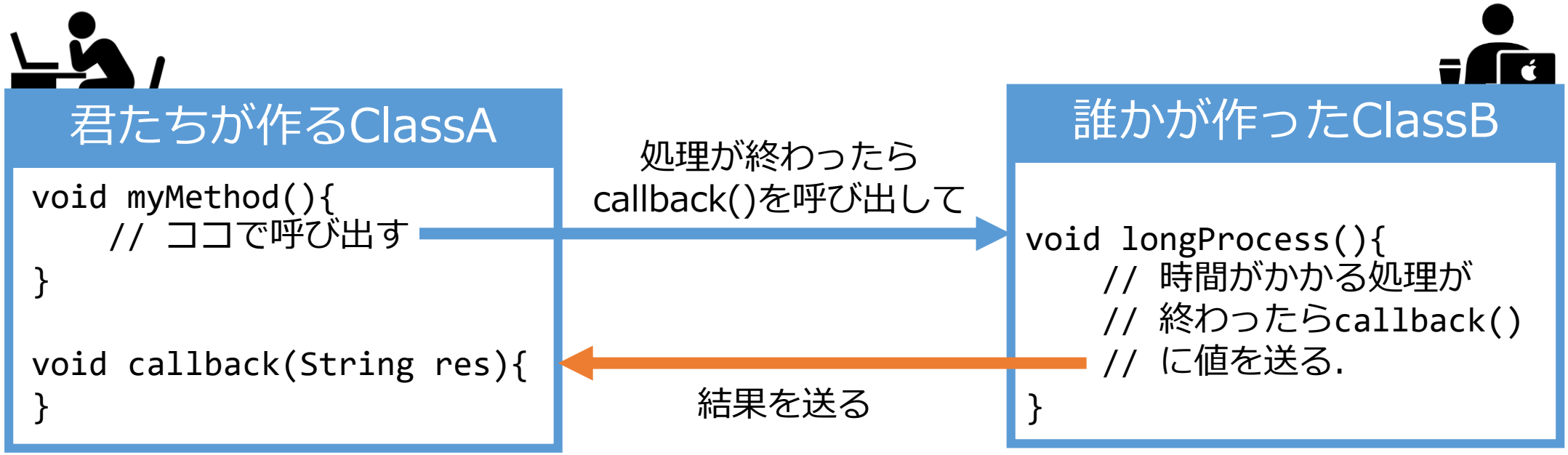
本ページの内容は
P22とほぼ同じ

- 複数の子クラスに同じメソッドを実装させることができる（抽象クラスと同様）.
- インタフェースがimplementsされているということは，**インタフェースに定義されているメソッドを必ず持っている**ことが保証される.
- インタフェースは**複数同時にimplementsできる**.

利点２があることによって，**コールバック**を実装することができる.

インタフェース
コールバックとしての使い方

コールバック
あるクラスに自分のメソッドを教えておき，処理が終わったら実行してもらうこと



インタフェース
コールバックとしての使い方

君たちが作るClassA

```
void myMethod(){
  ClassB b = new ClassB();
  res = b.longProcessA();
  // 後の処理は待たされる

  b.longProcessB();
  // 後の処理が実行できる
}

void callback(String res){
}
```

今までは，戻り値で結果をそのまま返していた

結果出次第コールバックという処理が実現できる

誰かが作ったClassB

```
void longProcessA(){
  // 時間がかかる処理
  // を実行した後
  // 戻り値として返す
}
void longProcessB(){
  // 時間がかかる処理を
  // マルチスレッドで実行後
  // callback()に値を送る
}
```

インタフェース
コールバックとしての使い方

君たちが作るClassA

```
void myMethod(){
  ClassB b = new ClassB();
  b.longProcess(this);
  // 後の処理が実行できる
}

void callback(String res){
}
```

thisを渡してあげる

ClassAインスタンスからcallback()を呼び出せる

誰かが作ったClassB

```
void longProcess(ClassA a){
  // 時間がかかる処理を
  // マルチスレッドで実施
  // インスタンスaのcallback()
  a.callback(res);
}
```

ClassAのインスタンス（あっちから見たthis）を受け取る

なぜClassBの作者がClassAやcallback()の存在を知っている？

ClassBは過去に開発されている→ClassAが将来開発されるかはわからない.
ClassCで同じ処理を依頼したいときにClassBはClassA専用に開発されていることになる.

インタフェース
コールバックとしての使い方

interface

```
public interface MyCallback{
  void callback(String res);
}
```

ClassBと同時に準備

callback()を依頼するために
MyCallbackを実装する.

Implements MyCallback

君たちが作るClassA

```
void myMethod(){
  ClassB b = new ClassB();
  b.longProcess(this);
  // 後の処理が実行できる
}

void callback(String res){
}
```

誰かが作ったClassB

```
void longProcess(MyCallback c){
  // 時間がかかる処理を
  // マルチスレッドで実施
  // インスタンスcのcallback()
  c.callback(res);
}
```

callback()を
呼び出せる

ClassAのインスタンス
（あっちから見たthis）
だが，**MyCallback型で受け取る**

ClassBの作者はClassAだかClassCだか知らないけれど,
MyCallbackを実装してくれているのでcallback()を呼び出せる.

MyCallbackを実装している＝callback()が実装されていることが保証されている.

インタフェース
コールバックとしての使い方

コールバックを用いることで

- Web通信等，長時間の処理が想定されるものをマルチスレッド実装し，処理が完了した際にメソッドを呼び出してもらう事が可能になる.
- ボタンクリックイベント等，発生タイミングが予測できない状況で，イベント発生時にメソッドを呼び出してもらうことが可能になる.

インタフェース
コールバックとしての使い方

コールバックは見方を変えると，**別のクラスに自分が実装したメソッドを渡す方法**ともいえる．

君たちが作るClassA

```
void myMethod(){
  ClassB b = new ClassB();
  b.longProcessA(???);

  b.longProcessB(this);
}
void callback(String res){
}
```

引数にメソッドを指定できないのでcallback()を送ることができない

引数thisでClassAのインスタンスを渡す

ClassAのメソッドを実行

誰かが作ったClassB

```
void longProcessA(???){
  // 実装方法がない
}
void longProcessB(ClassA a){
  // ClassAのメソッドが実行
  // できる.
  a.callback();
}
```

インタフェース
コールバックとしての使い方

コールバックは見方を変えると，**別のクラスに自分が実装したメソッドを渡す方法**ともいえる．

Implements MyCallback

君たちが作るClassA

```
void myMethod(){
  ClassB b = new ClassB();
  b.longProcessA(???);

  b.longProcessB(this);
}
void callback(String res){
}
```

引数で**MyCallbackを実装しているクラスのインスタンス**を渡す

callback()を実行

誰かが作ったClassB

```
void longProcessA(???){
  // 実装方法がない
}
void longProcessB(MyCallback c){
  // MyCallbackのメソッドが実行
  // できる(受け取りたい処理を
  // callback()内に実装させる)
  c.callback();
}
```

インタフェース
コールバックとしての使い方

コールバックでメソッドを送ることで

- 一定間隔で実行したい処理（TimerTask）の実装を標準API（Timer）に投げることができる.（TimerTaskは抽象クラス）
- 並列処理（Runnable）の実装を標準API（Thread）に投げることができる.（Runnableはインタフェース）

次週予告

※次週以降も計算機室

前半

Javaの三大要素，最後の砦，
ポリモーフィズム（多態性）について学ぶ.

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．．．）